python™

Lecture 4:

Data Manipulation & Visualization with Pandas

Section 4.1 | Data wrangling, combine & reshape

Section 4.1 | Data wrangling, combine & reshape

4.1.1: Hierarchical Indexing

>> 4.1.1: Hierarchical Indexing

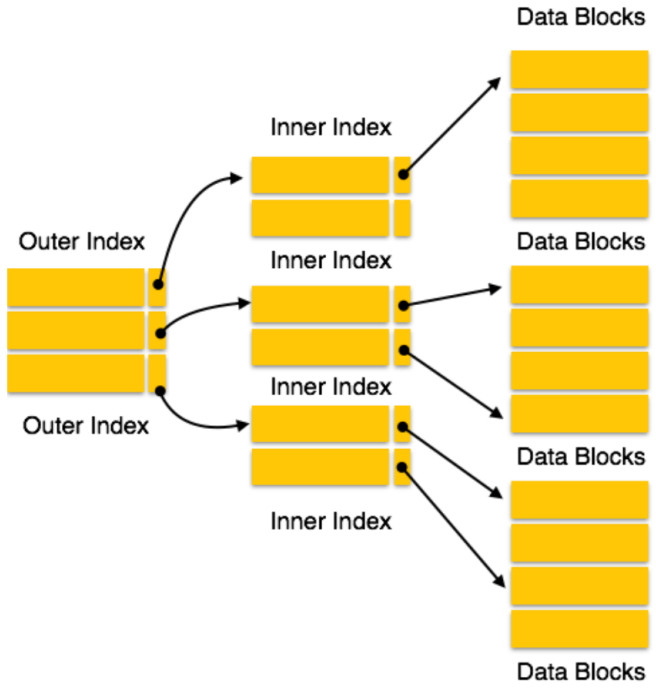
```
import numpy as np
import pandas as pd
```

```
data = pd.DataFrame(np.random.randint(1,10, size=(6,9)),
                    index=[['A', 'B', 'B', 'C', 'C', 'C'],
                           [1, 2, 3, 4, 5, 6]],
                    columns=[['I', 'I', 'II', 'II', 'II', 'II', 'III', 'III', 'III'],
                           [1, 2, 1, 2, 3, 4, 1, 2, 3]])
data
```

		I		II				III		
		1	2	1	2	3	4	1	2	3
A	1	7	4	6	4	7	6	1	5	5
B	2	3	6	5	4	6	3	6	7	1
	3	6	5	8	7	1	2	8	5	1
C	4	8	5	1	4	9	6	6	3	7
	5	5	8	7	1	3	5	1	2	8
	6	1	6	8	2	3	1	5	8	5

```
data.index
```

```
MultiIndex([('A', 1),
            ('B', 2),
            ('B', 3),
            ('C', 4),
            ('C', 5),
            ('C', 6)],
           )
```



- Multiple Index levels on one axis
- A “MultiIndex” object as index
- The “gap” means “use the same label above” (like in Excel)

Horizontal Selection

Select column

```
data['II']
```

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

Select column range

```
data.loc[:,'I':'II']
```

		I		II			
		1	2	1	2	3	4
A	1	7	4	6	4	7	6
B	2	3	6	5	4	6	3
	3	6	5	8	7	1	2
C	4	8	5	1	4	9	6
	5	5	8	7	1	3	5
	6	1	6	8	2	3	1

Select specific columns

```
data.loc[:, ['I','III']]
```

		I		III		
		1	2	1	2	3
A	1	7	4	1	5	5
B	2	3	6	6	7	1
	3	6	5	8	5	1
C	4	8	5	6	3	7
	5	5	8	1	2	8
	6	1	6	5	8	5

Select column inner level

```
data['II'][3]
```

```
A  1    7
B  2    6
   3    1
C  4    9
   5    3
   6    3
Name: 3, dtype: int32
```

Vertical Selection

Select outer index

```
data.loc['B',:]
```

	I		II				III		
	1	2	1	2	3	4	1	2	3
2	3	6	5	4	6	3	6	7	1
3	6	5	8	7	1	2	8	5	1

Select outer index range

```
data.loc['B':'C',:]
```

		I		II				III		
		1	2	1	2	3	4	1	2	3
B	2	3	6	5	4	6	3	6	7	1
	3	6	5	8	7	1	2	8	5	1
C	4	8	5	1	4	9	6	6	3	7
	5	5	8	7	1	3	5	1	2	8
	6	1	6	8	2	3	1	5	8	5

Select specific outer index

```
data.loc[['A','C'],:]
```

	I		II				III		
	1	2	1	2	3	4	1	2	3
A	1	7	4	6	4	7	6	1	5	5
C	4	8	5	1	4	9	6	6	3	7
	5	5	8	7	1	3	5	1	2	8
	6	1	6	8	2	3	1	5	8	5

Select inner index

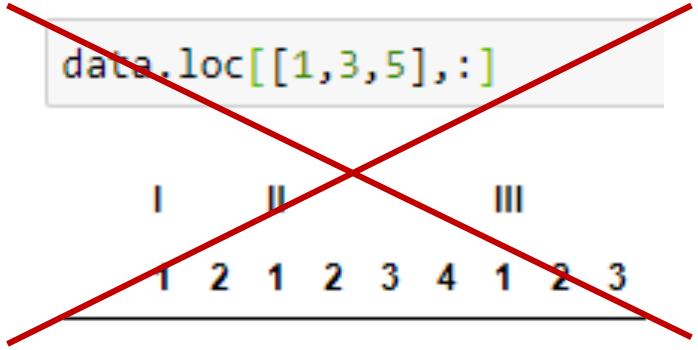
```
data.loc['B',:].loc[2,:]
```

```
I    1    3
     2    6
II   1    5
     2    4
     3    6
     4    3
III  1    6
     2    7
     3    1
Name: 2, dtype: int32
```

Level Selection

>> Select all data that has level 2 index equal to (1, 3 and 5)

```
data.loc[[1,3,5],:]
```




```
idx = pd.IndexSlice
data.loc[idx[:,[1,3,5]],:]
```

		I		II			III		
		1	2	1	2	3	4	1	2	3
A	1	7	4	6	4	7	6	1	5	5
B	3	6	5	8	7	1	2	8	5	1
C	5	5	8	7	1	3	5	1	2	8

>> Exercise: Select all data that has level 2 columns equal to 1

Cross-Section Selection

Assign index name

```
data.index.names = ['one', 'two']
data.columns.names = ['first', 'second']
data
```

first		I		II			III		
second		1	2	1	2	3	4	1	2	3
one	two									
A	1	7	4	6	4	7	6	1	5	5
B	2	3	6	5	4	6	3	6	7	1
	3	6	5	8	7	1	2	8	5	1
C	4	8	5	1	4	9	6	6	3	7
	5	5	8	7	1	3	5	1	2	8
	6	1	6	8	2	3	1	5	8	5

One level depth selection

```
data.xs(1, level='second', axis=1)
```

first		I	II	III
one	two			
A	1	7	6	1
B	2	3	5	6
	3	6	8	8
C	4	8	1	6
	5	5	7	1
	6	1	8	5

Multiple level depth selection

```
data.xs(('I', 2), level=('first', 'second'), axis=1)
```

first		I
second		2
one	two	
A	1	4
B	2	6
	3	5
C	4	5
	5	8
	6	6

Section 4.1 | Data wrangling, combine & reshape

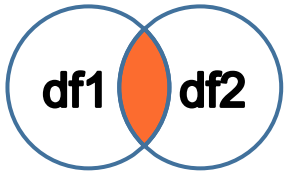
4.1.2: Combining and Merging

>> 4.1.2: Combining and Merging – Horizontal Joining

df1	key	A	B
0	*K0*	A0	B0
1	*K1*	A1	B1
2	*K2*	A2	B2
3	*K3*	A3	B3

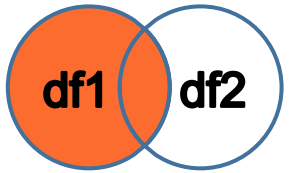
+

df2	key	C	D
0	*K0*	C0	D0
1	*K1*	C1	D1
2	*K1*	C2	D2
3	*K4*	C3	D3



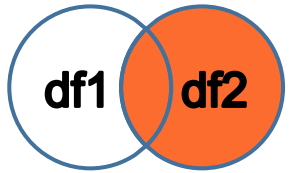
INNER JOIN

INNER JOIN	key	A	B	C	D
0	*K0*	A0	B0	C0	D0
1	*K1*	A1	B1	C1	D1
2	*K1*	A1	B1	C2	D2



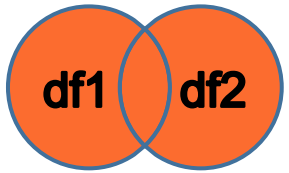
LEFT OUTER JOIN

LEFT JOIN	key	A	B	C	D
0	*K0*	A0	B0	C0	D0
1	*K1*	A1	B1	C1	D1
2	*K1*	A1	B1	C2	D2
3	*K2*	A2	B2	NaN	NaN
4	*K3*	A3	B3	NaN	NaN



RIGHT OUTER JOIN

RIGHT JOIN	key	A	B	C	D
0	*K0*	A0	B0	C0	D0
1	*K1*	A1	B1	C1	D1
2	*K1*	A1	B1	C2	D2
3	*K4*	NaN	NaN	C3	D3



FULL OUTER JOIN

OUTER JOIN	key	A	B	C	D
0	*K0*	A0	B0	C0	D0
1	*K1*	A1	B1	C1	D1
2	*K1*	A1	B1	C2	D2
3	*K2*	A2	B2	NaN	NaN
4	*K3*	A3	B3	NaN	NaN
5	*K4*	NaN	NaN	C3	D3

>> 4.1.2: Combining and Merging – Horizontal Joining

```
df1 = pd.DataFrame({
    'key': ['K0', 'K1', 'K2', 'K3'],
    'A': ['A0', 'A1', 'A2', 'A3'],
    'B': ['B0', 'B1', 'B2', 'B3']
})
df1
```

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

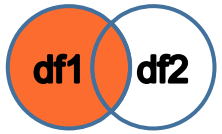
```
df2 = pd.DataFrame({
    'key': ['K0', 'K1', 'K1', 'K4'],
    'C': ['C0', 'C1', 'C2', 'C3'],
    'D': ['D0', 'D1', 'D2', 'D3']
})
df2
```

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K1	C2	D2
3	K4	C3	D3

INNER JOIN

```
pd.merge(df1, df2, on='key')
# Same as pd.merge(df1, df2)
```

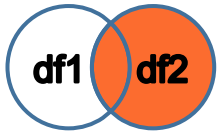
	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K1	A1	B1	C2	D2



LEFT OUTER JOIN

```
pd.merge(df1, df2, on='key', how='left')
```

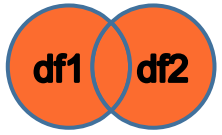
	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K1	A1	B1	C2	D2
3	K2	A2	B2	NaN	NaN
4	K3	A3	B3	NaN	NaN



RIGHT OUTER JOIN

```
pd.merge(df1, df2, on='key', how='right')
```

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K1	A1	B1	C2	D2
3	K4	NaN	NaN	C3	D3



FULL OUTER JOIN

```
pd.merge(df1, df2, on='key', how='outer')
```

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K1	A1	B1	C2	D2
3	K2	A2	B2	NaN	NaN
4	K3	A3	B3	NaN	NaN
5	K4	NaN	NaN	C3	D3

>> 4.1.2: Combining and Merging – Vertical Joining

df1	key	A	B
0	*K0*	A0	B0
1	*K1*	A1	B1
2	*K2*	A2	B2
3	*K3*	A3	B3

df2	key	C	D
0	*K0*	C0	D0
1	*K1*	C1	D1
2	*K1*	C2	D2
3	*K4*	C3	D3

	key	A	B	C	D
0	*K0*	A0	B0	NaN	NaN
1	*K1*	A1	B1	NaN	NaN
2	*K2*	A2	B2	NaN	NaN
3	*K3*	A3	B3	NaN	NaN
0	*K0*	NaN	NaN	C0	D0
1	*K1*	NaN	NaN	C1	D1
2	*K1*	NaN	NaN	C2	D2
3	*K4*	NaN	NaN	C3	D3

Vertical joining DataFrame with pd.concat()

```
pd.concat([df1, df2])
```

	key	A	B	C	D
0	K0	A0	B0	NaN	NaN
1	K1	A1	B1	NaN	NaN
2	K2	A2	B2	NaN	NaN
3	K3	A3	B3	NaN	NaN
0	K0	NaN	NaN	C0	D0
1	K1	NaN	NaN	C1	D1
2	K1	NaN	NaN	C2	D2
3	K4	NaN	NaN	C3	D3

Can also Horizontal joining on index

```
pd.concat([df1, df2], axis=1)
```

	key	A	B	key	C	D
0	K0	A0	B0	K0	C0	D0
1	K1	A1	B1	K1	C1	D1
2	K2	A2	B2	K1	C2	D2
3	K3	A3	B3	K4	C3	D3

Section 4.1 | Data wrangling, combine & reshape

4.1.3: Reshaping Data

>> 4.1.3: Reshaping Data – What?

1	2	3	4	5

Row to column transformation

1
2
3
4
5

OR

col1	col2	col3
0	red	a
1	yellow	b
2	blue	c
3	green	d

Long data to Wide data

col1	red	yellow	blue	green
1	a	b	NaN	NaN
2	NaN	NaN	c	d

>> 4.1.3: Reshaping Data – How?

>> Reshaping with Hierarchical Indexing

```
slice1 = data.loc[idx[:'B'],'II']
slice1
```

	second	1	2	3	4
one	two				
A	1	7	8	2	7
B	2	7	5	1	6
	3	5	4	7	2

>> Row to Column transformation with **unstack()** method

```
slice1.unstack()
```

second	1			2			3			4		
two	1	2	3	1	2	3	1	2	3	1	2	3
one												
A	7.0	NaN	NaN	8.0	NaN	NaN	2.0	NaN	NaN	7.0	NaN	NaN
B	NaN	7.0	5.0	NaN	5.0	4.0	NaN	1.0	7.0	NaN	6.0	2.0

>> Column to Row transformation with **stack()** method

```
slice1.stack()
```

```
one  two  second
A    1    1         7
               2         8
               3         2
               4         7
B    2    1         7
               2         5
               3         1
               4         6
     3    1         5
               2         4
               3         7
               4         2
dtype: int32
```

>> By default the inner most index level got affected (unstack or stack)

>> You can unstack (or stack) specific level by passing the level (numer or name) as argument

```
slice1.unstack(0)
```

second	1		2		3		4	
one	A	B	A	B	A	B	A	B
two								
1	7.0	NaN	8.0	NaN	2.0	NaN	7.0	NaN
2	NaN	7.0	NaN	5.0	NaN	1.0	NaN	6.0
3	NaN	5.0	NaN	4.0	NaN	7.0	NaN	2.0

Section 4.1 | Data wrangling, combine & reshape

4.1.4: Pivoting

THE POWER OF PIVOTING

```
df = pd.DataFrame({
    'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
    'baz': [1, 2, 3, 4, 5, 6],
    'zoo': ['x', 'y', 'z', 'q', 'w', 't']
})
df
```

```
df.pivot(index='foo', columns='bar', values='baz')
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

>> 4.1.3: Pivoting – Let's get started

MAGIC CODE INSTITUTE
Leverage your tech skills

>> Load the 'Macrodata' dataset

```
data = pd.read_csv('macrodata.csv')
data
```

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	m1	tbilrate	unemp	pop	infl	realint
0	1959.0	1.0	2710.349	1707.4	286.898	470.045	1886.9	28.980	139.7	2.82	5.8	177.146	0.00	0.00
1	1959.0	2.0	2778.801	1733.7	310.859	481.301	1919.7	29.150	141.7	3.08	5.1	177.830	2.34	0.74
2	1959.0	3.0	2775.488	1751.8	289.226	491.260	1916.4	29.350	140.5	3.82	5.3	178.657	2.74	1.09
3	1959.0	4.0	2785.204	1753.7	299.356	484.052	1931.3	29.370	140.0	4.33	5.6	179.386	0.27	4.06
4	1960.0	1.0	2847.699	1770.5	331.722	462.199	1955.5	29.540	139.6	3.50	5.2	180.007	2.31	1.19
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
198	2008.0	3.0	13324.600	9267.7	1990.693	991.551	9838.3	216.889	1474.7	1.17	6.0	305.270	-3.16	4.33
199	2008.0	4.0	13141.920	9195.3	1857.661	1007.273	9920.4	212.174	1576.5	0.12	6.9	305.952	-8.79	8.91
200	2009.0	1.0	12925.410	9209.2	1558.494	996.287	9926.4	212.671	1592.8	0.22	8.1	306.547	0.94	-0.71
201	2009.0	2.0	12901.504	9189.0	1456.678	1023.528	10077.5	214.469	1653.6	0.18	9.2	307.226	3.37	-3.19
202	2009.0	3.0	12990.341	9256.0	1486.398	1044.088	10040.6	216.385	1673.9	0.12	9.6	308.013	3.56	-3.44

203 rows × 14 columns

>> 4.1.3: Pivoting

>> Take out a slice of the data the '*cool*' way

```
# Construct an Index object with specific columns
columns = pd.Index(['realgdp', 'infl', 'unemp'], name='stat')
columns
```

```
Index(['realgdp', 'infl', 'unemp'], dtype='object', name='stat')
```

```
# Take a slice of data by reindex the DataFrame with column index
# we just created
new_data = data.reindex(columns=columns)
new_data
```

stat	realgdp	infl	unemp
0	2710.349	0.00	5.8
1	2778.801	2.34	5.1
2	2775.488	2.74	5.3
3	2785.204	0.27	5.6
4	2847.699	2.31	5.2
...	...	...	...
198	13324.600	-3.16	6.0
199	13141.920	-8.79	6.9
200	12925.410	0.94	8.1
201	12901.504	3.37	9.2
202	12990.341	3.56	9.6

>> Construct new time Index with '*year*' and '*quarter*'

```
# Combine 'year' and 'quarter' columns to create new Combined Index
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter, name='date')
periods
```

```
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', name='date', length=203, freq='Q-DEC')
```

```
# Convert quarterly index to timestamp index
periods.end_time
```

```
DatetimeIndex(['1959-03-31 23:59:59.999999999',
               '1959-06-30 23:59:59.999999999',
               '1959-09-30 23:59:59.999999999',
               '1959-12-31 23:59:59.999999999',
               '1960-03-31 23:59:59.999999999',
               '1960-06-30 23:59:59.999999999',
               '1960-09-30 23:59:59.999999999',
               '1960-12-31 23:59:59.999999999',
               '1961-03-31 23:59:59.999999999',
               '1961-06-30 23:59:59.999999999',
```

>> 4.1.3: Pivoting

MAGIC CODE INSTITUTE

>> Assign new index to new data

```
new_data.index = periods.end_time
new_data
```

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

203 rows × 3 columns

>> Let's play around a little bit

```
new_data.stack()
```

```
date                           stat
1959-03-31 23:59:59.999999999  realgdp     2710.349
                               infl           0.000
                               unemp          5.800
1959-06-30 23:59:59.999999999  realgdp     2778.801
                               infl           2.340
                                             ...
2009-06-30 23:59:59.999999999  infl           3.370
                               unemp          9.200
2009-09-30 23:59:59.999999999  realgdp    12990.341
                               infl           3.560
                               unemp          9.600
Length: 609, dtype: float64
```

```
new_data.stack().reset_index()
```

	date	stat	0
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
...	...	...	...

>> 4.1.3: Pivoting

MAGIC CODE INSTITUTE

>> Rename DataFrame columns

```
# Rename DataFrame columns
stacked = new_data.stack().reset_index().rename(columns={0: 'value'})
stacked
```

	date	stat	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
...	...	...	...
604	2009-06-30 23:59:59.999999999	infl	3.370
605	2009-06-30 23:59:59.999999999	unemp	9.200
606	2009-09-30 23:59:59.999999999	realgdp	12990.341
607	2009-09-30 23:59:59.999999999	infl	3.560
608	2009-09-30 23:59:59.999999999	unemp	9.600

609 rows × 3 columns

>> And pivot it back to before transformation

```
# Unstack data by using pivot method
pivoted = stacked.pivot(index='date', columns='stat', values='value')
pivoted
```

stat	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

>> HASTA LA VISTA! WE WILL BE BACK!

Section 4.1 | Plotting and Visualization with Matplotlib & Seaborn

Section 4.2 | Data wrangling, combine & reshape

4.2.1: Matplotlib Primer

>> 4.2.1: Matplotlib Primer

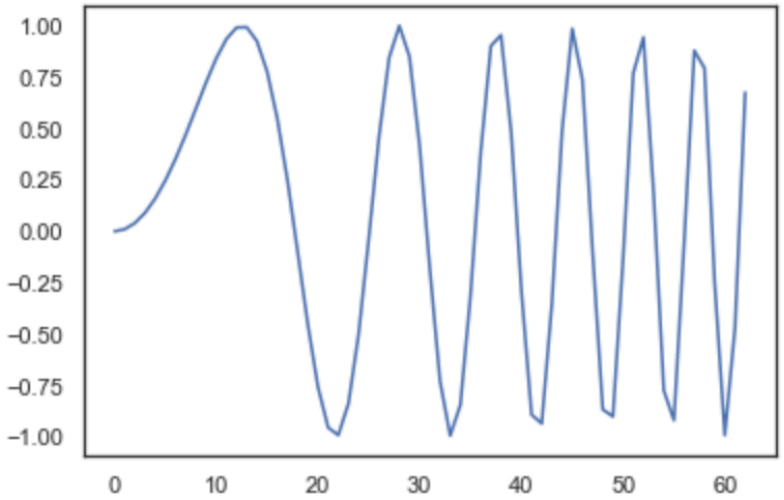
>> Today our new magic words is

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

>> And our first plot

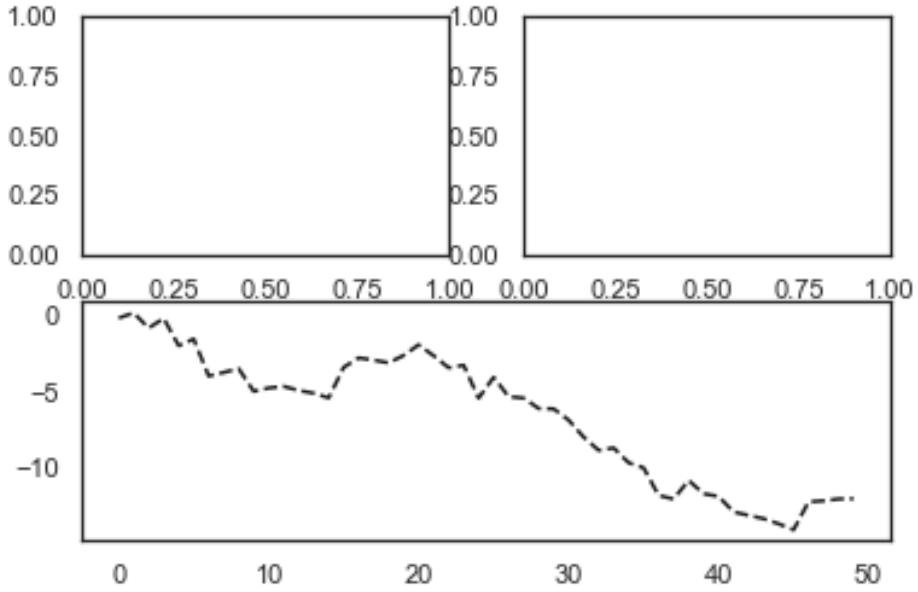
```
import math
data = np.sin(np.arange(0, 2*math.pi, 0.1)**2)
```

```
fig = plt.plot(data)
```



>> The correct way is to create a figure with *plt.figure()*
>> Adding subplots is EZPZ

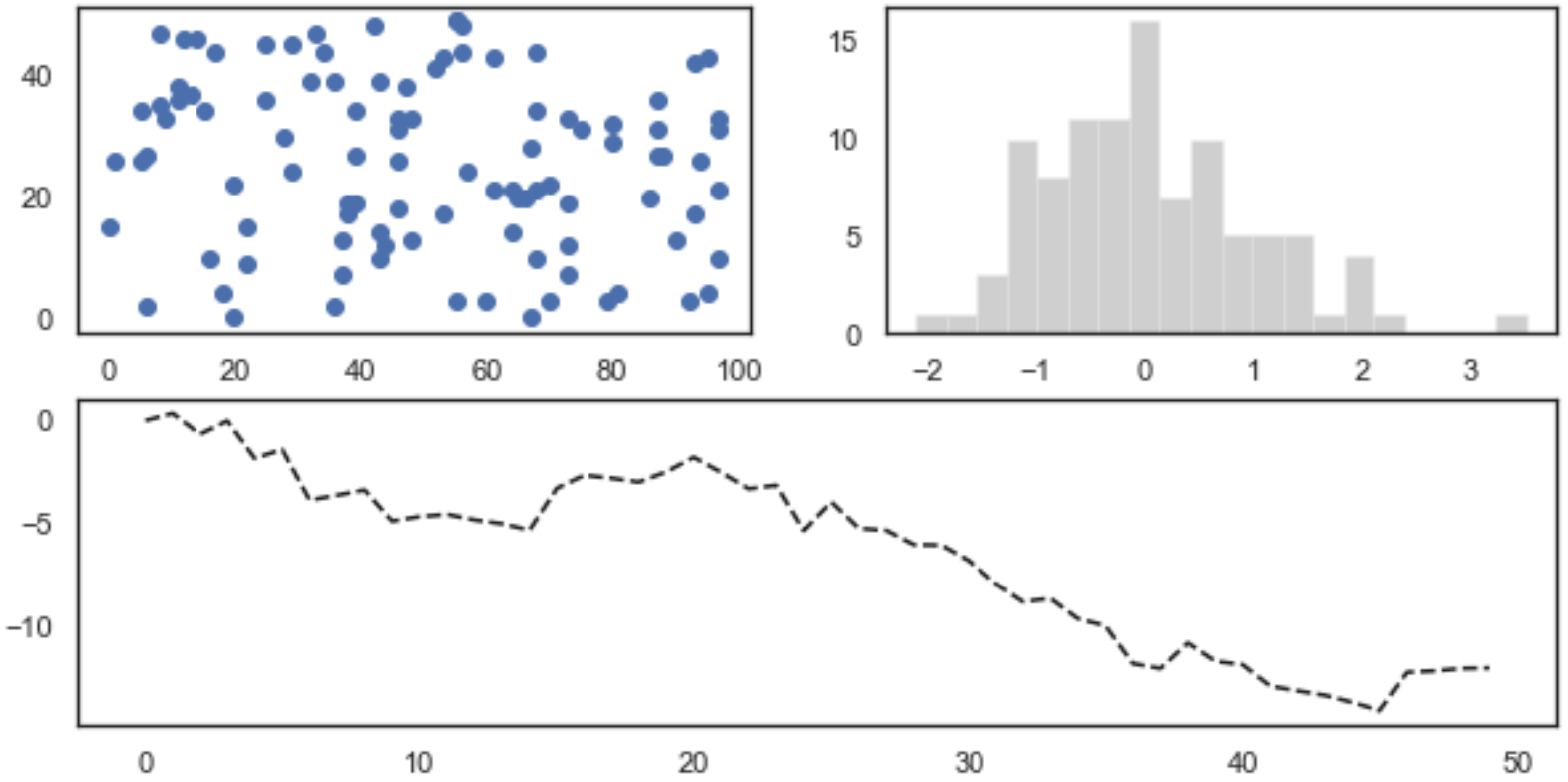
```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, (3, 4))
plt.plot(np.random.randn(50).cumsum(), 'k--')
plt.show()
```



>> 4.2.1: Matplotlib Primer

>> And even more subplots. No problemo!

```
ax1.scatter(np.random.randint(0, 100, 100), np.random.randint(0, 50, 100))
ax2.hist(np.random.randn(100), bins=20, color='k', alpha=0.2)
fig.set_size_inches(10, 5)
fig
```

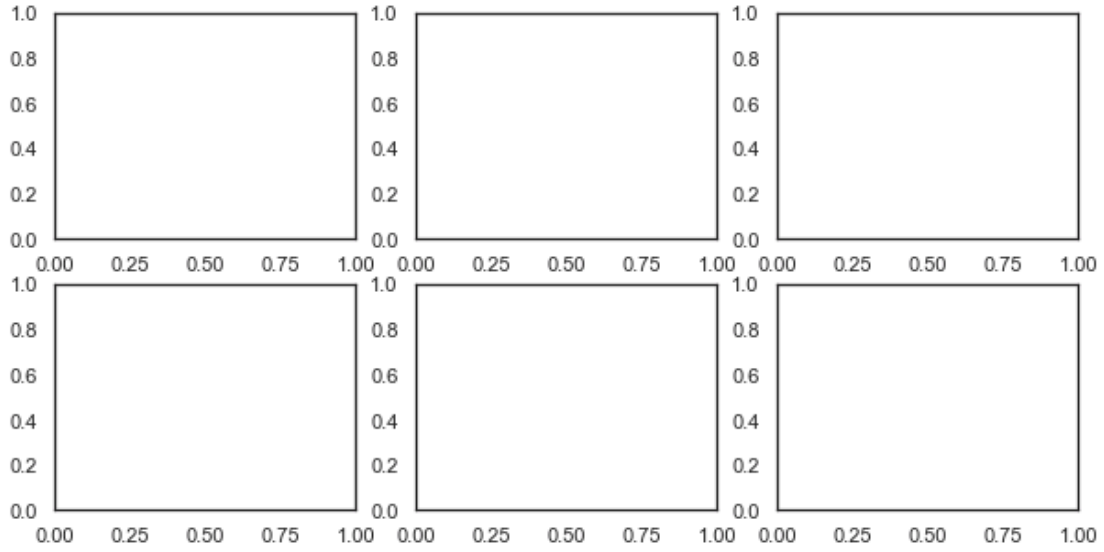


>> 4.2.1: Matplotlib Primer

>> Quicky create a grid of subplots. The axes object can be accessed like ndarray such as *axes[0,1]*

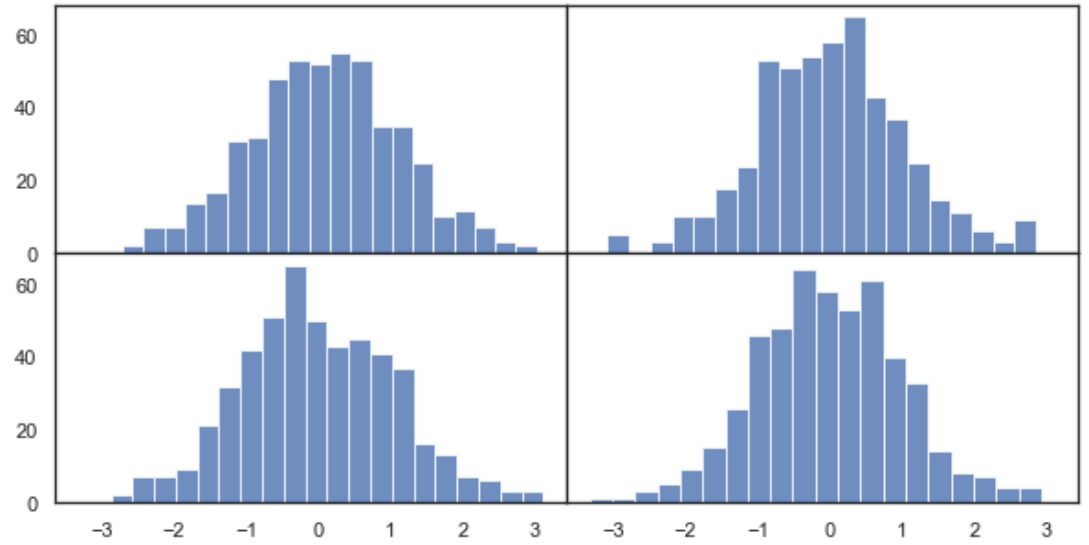
```
# Quickly create subplots
fig, axes = plt.subplots(2, 3, figsize=(10, 5))
axes
```

```
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```



>> Access the grid of subplots and fill them with data

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(10,5))
for i in range(axes.shape[0]):
    for j in range(axes.shape[1]):
        data = np.random.randn(500)
        axes[i,j].hist(data, bins=20, color='b', alpha=0.8)
plt.subplots_adjust(wspace=0, hspace=0)
```

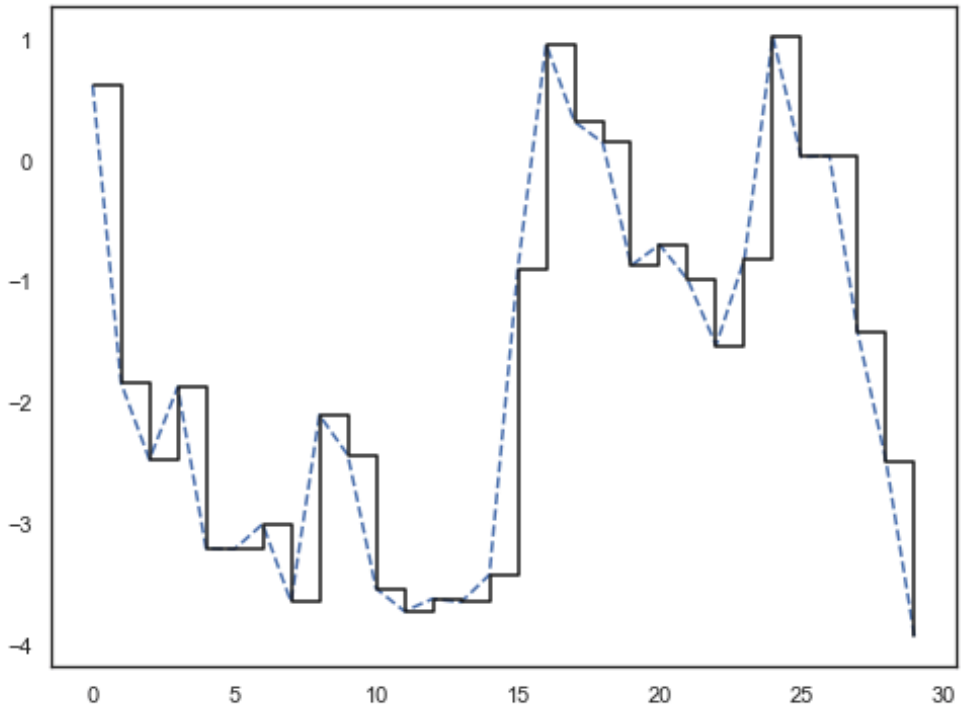


>> 4.2.1: Matplotlib Primer

>> How about ploting multiple data on the same subplot?

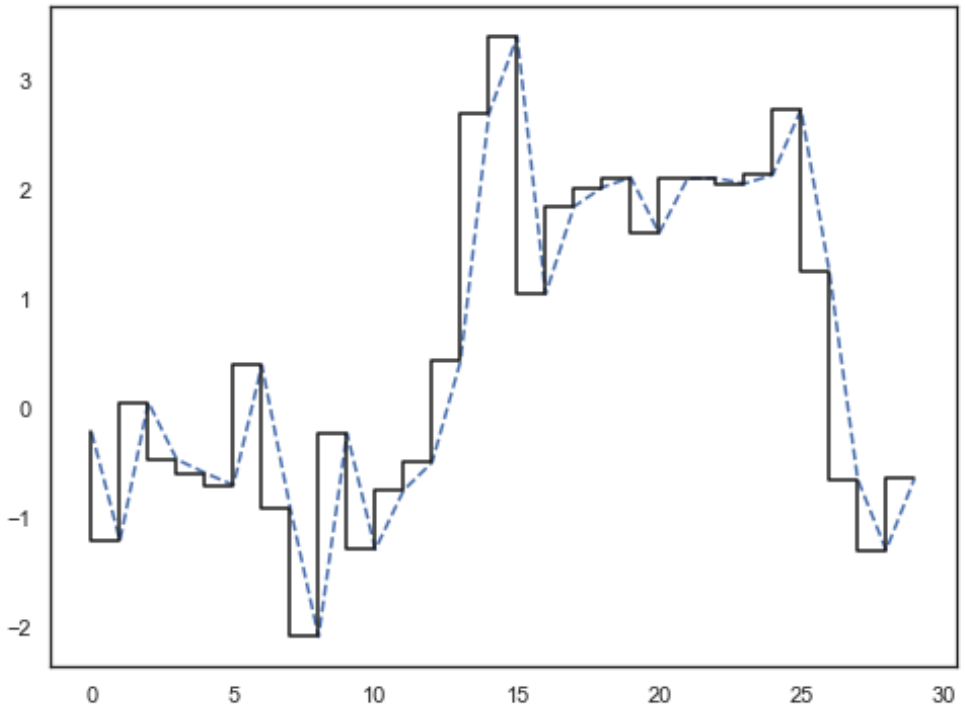
```
data = np.random.randn(30).cumsum()
fig = plt.figure(figsize=(8,6))
plt.plot(data, 'b--', label='default')
plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
```

```
[<matplotlib.lines.Line2D at 0x1293e2460>]
```




```
data = np.random.randn(30).cumsum()
fig = plt.figure(figsize=(8,6))
plt.plot(data, 'b--', label='default')
plt.plot(data, 'k-', drawstyle='steps-pre', label='steps-pre')
```

```
[<matplotlib.lines.Line2D at 0x12a6c9c70>]
```



>> 4.2.1: Matplotlib Primer

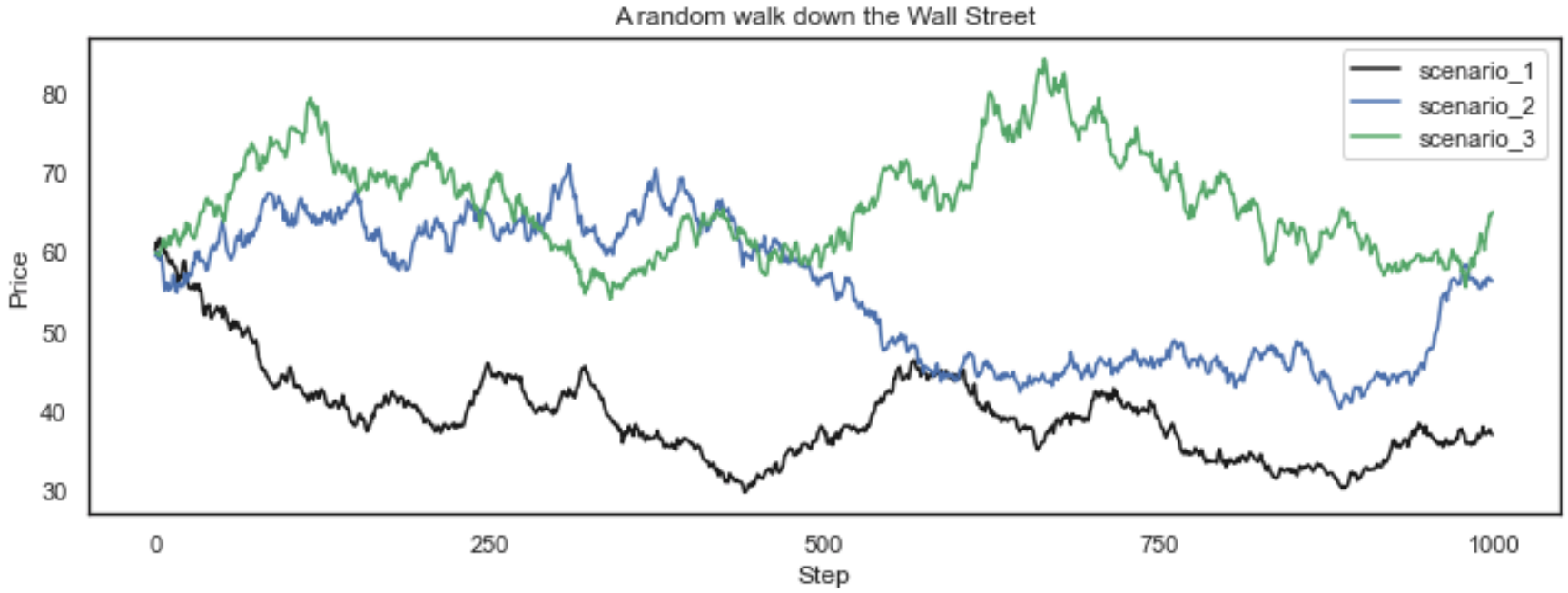
>> Let’s put a name on everything. Do we? Set label and title of the plot

```
# Set title and label for plot
current_price = 60
simulated = 60*np.exp(np.random.normal(size=1000)*0.0125).cumprod()
fig = plt.figure(figsize=(12,4))
ax = fig.add_subplot(1, 1, 1)
ax.plot(simulated, 'k-', label='scenario_1')
ax.set_xticks(list(range(0, 1250, 250)))
ax.set_title('A random walk down the Wall Street')
ax.set_xlabel('Step')
ax.set_ylabel('Price')
```

```
Text(0, 0.5, 'Price')
```

>> Set and display legend for each dataset

```
def simulate_price(current_price):
    data = current_price*np.exp(np.random.normal(size=1000)*0.0125).cumprod()
    return data
ax.plot(simulate_price(60), 'b-', label='scenario_2')
ax.plot(simulate_price(60), 'g-', label='scenario_3')
ax.legend(loc='best')
fig
```

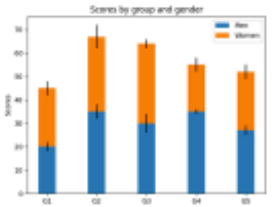


>> 4.2.1: Matplotlib Primer

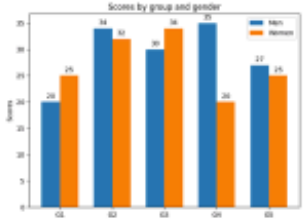
>> MATPLOTLIB GALLERY

https://matplotlib.org/gallery/index.html

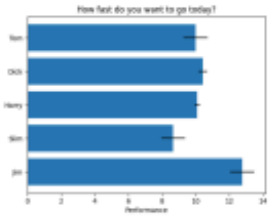
Check it out



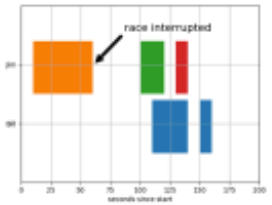
Stacked bar chart



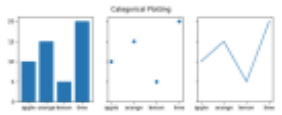
Grouped bar chart with labels



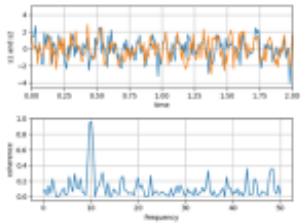
Horizontal bar chart



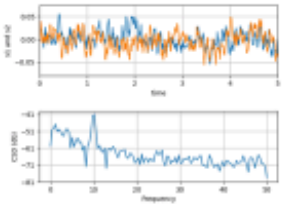
Broken Barh



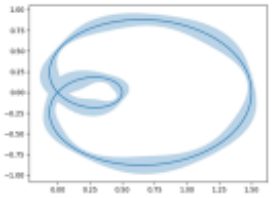
Plotting categorical variables



Plotting the coherence of two signals



CSD Demo



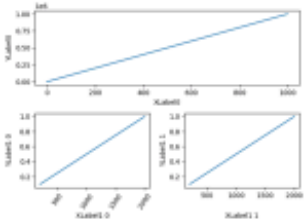
Curve with error band

>> 4.2.1: Matplotlib Primer

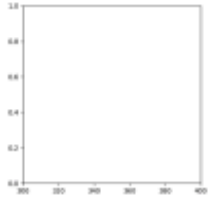
>> MATPLOTLIB GALLERY

https://matplotlib.org/gallery/index.html

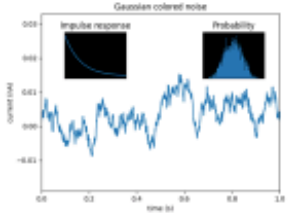
Check it out



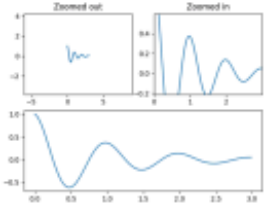
Aligning Labels



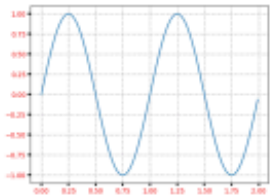
Axes box aspect



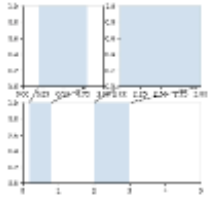
Axes Demo



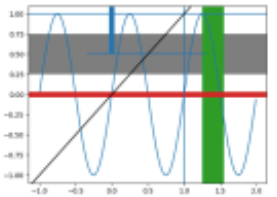
Controlling view limits using margins and sticky_edges



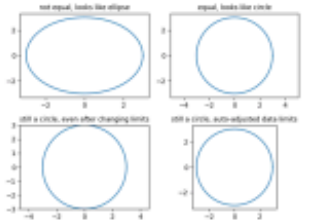
Axes Props



Axes Zoom Effect



axhspan Demo



Axis Equal Demo

Section 4.2 | Data wrangling, combine & reshape

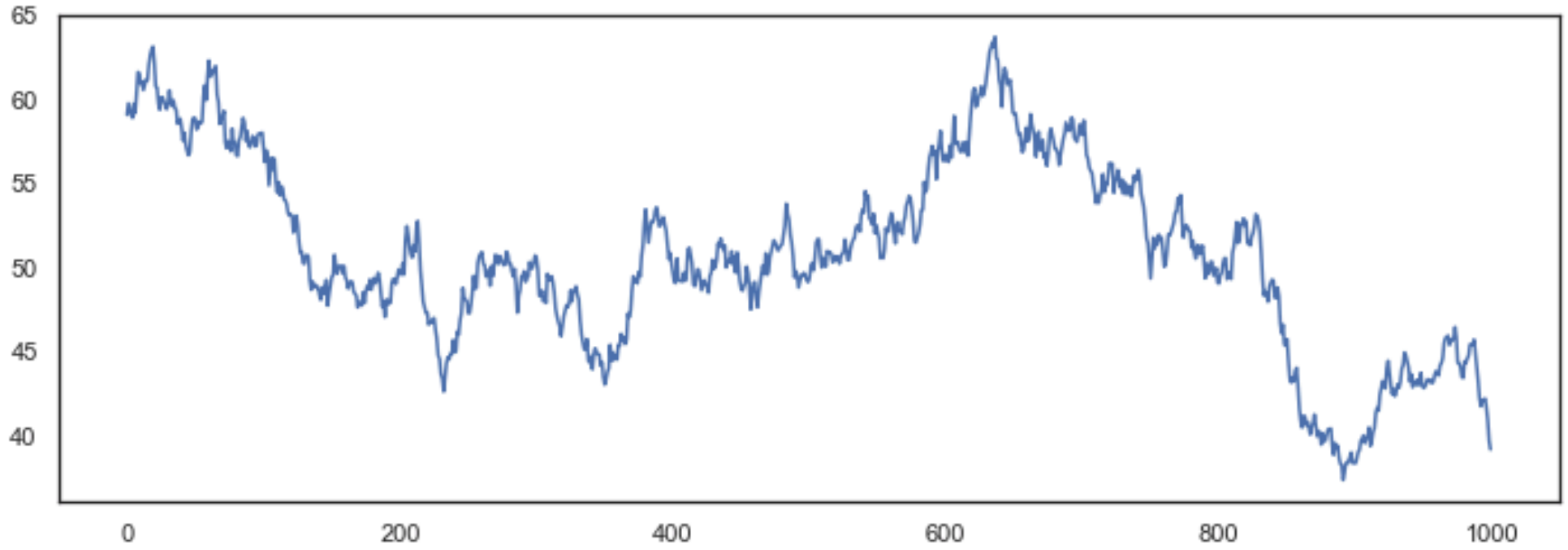
4.2.2: Plotting with Pandas and Seaborn

>> 4.2.2: Plotting with Pandas and Seaborn

>> Pandas also has Matplotlib API integrated to plot data

```
data = pd.Series(simulate_price(60))
data.plot(figsize=(12,4))
```

```
<AxesSubplot:>
```

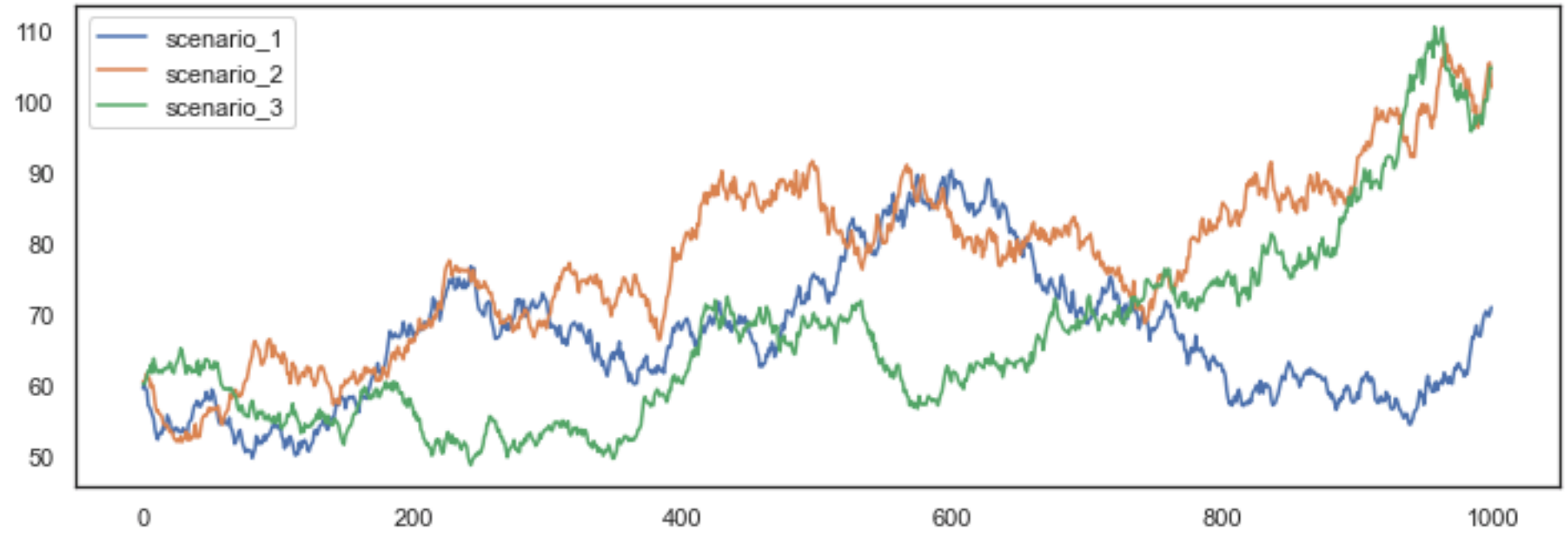


>> 4.2.2: Plotting with Pandas and Seaborn

>> The example that simulate 3 scenarios of stock price and then plot them, we can do it with ***pandas*** and in a ***much faster*** way. *You're quite welcomed!*

```
df = pd.DataFrame(
    np.transpose(np.array([simulate_price(60) for i in range(3)])),
    columns=['scenario_1', 'scenario_2', 'scenario_3']
)
df.plot(figsize=(12,4))
```

```
<AxesSubplot:>
```

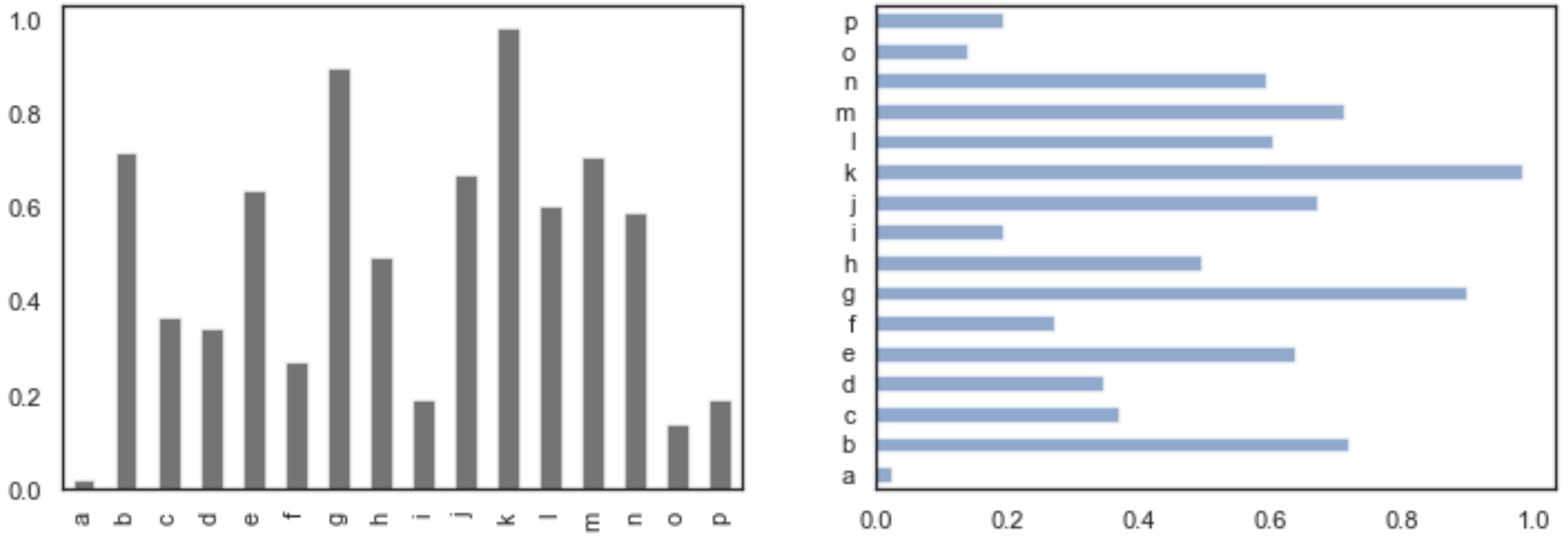


>> 4.2.2: Plotting with Pandas and Seaborn

>> Pandas can also do bar plot: vertical and horizontal

```
import string
fig, axs = plt.subplots(1, 2, figsize=(12,4))
data = pd.Series(np.random.rand(16), index=list(string.ascii_lowercase[:16]))
data.plot.bar(ax=axs[0], color='k', alpha=0.6)
data.plot.barh(ax=axs[1], color='b', alpha=0.6)
```

```
<AxesSubplot:>
```

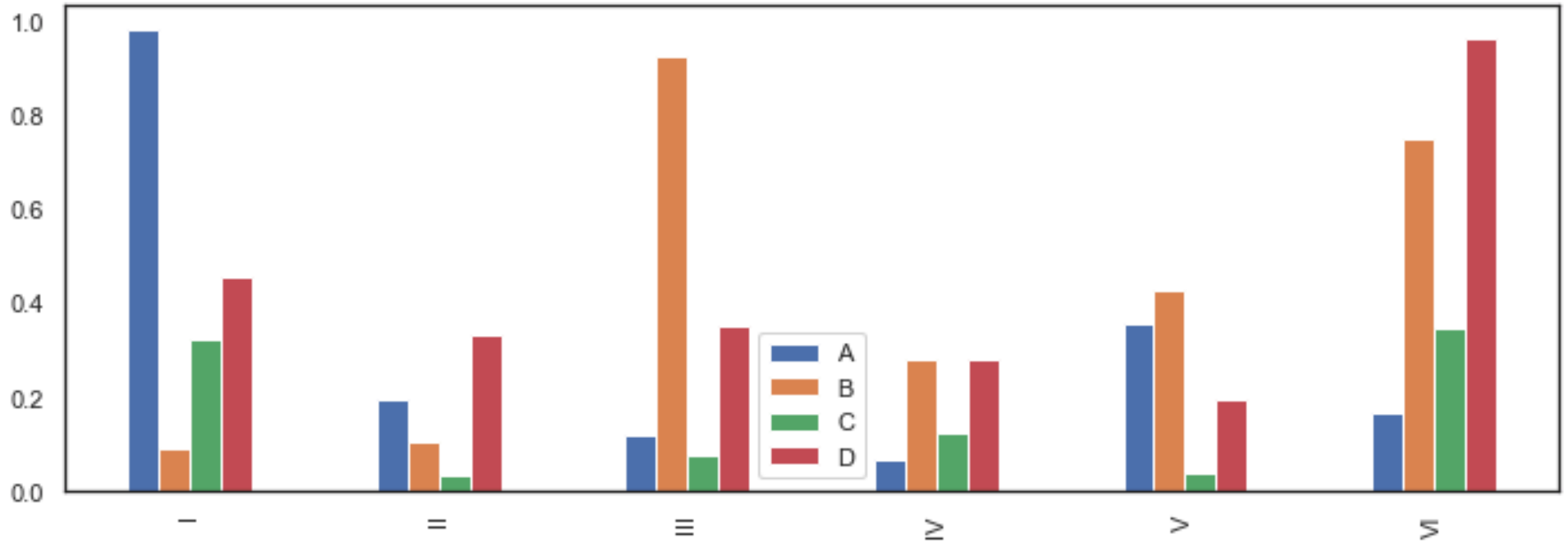


>> 4.2.2: Plotting with Pandas and Seaborn

>> And of course multiple bar plot and even more…

```
df = pd.DataFrame(np.random.rand(6, 4),
                  columns=['A', 'B', 'C', 'D'],
                  index = ['I', 'II', 'III', 'IV', 'V', 'VI'])
fig = df.plot.bar(figsize=(12,4))
fig.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x1294b3dc0>
```



>> 4.2.2: Plotting with Pandas and Seaborn

>> The seaborn package is built upon the matplotlib package and offer more sophisticated ways to present data. Let's explore!

>> Import the dataset

```
import seaborn as sns
tips = pd.read_csv('tips.csv')
tips
```

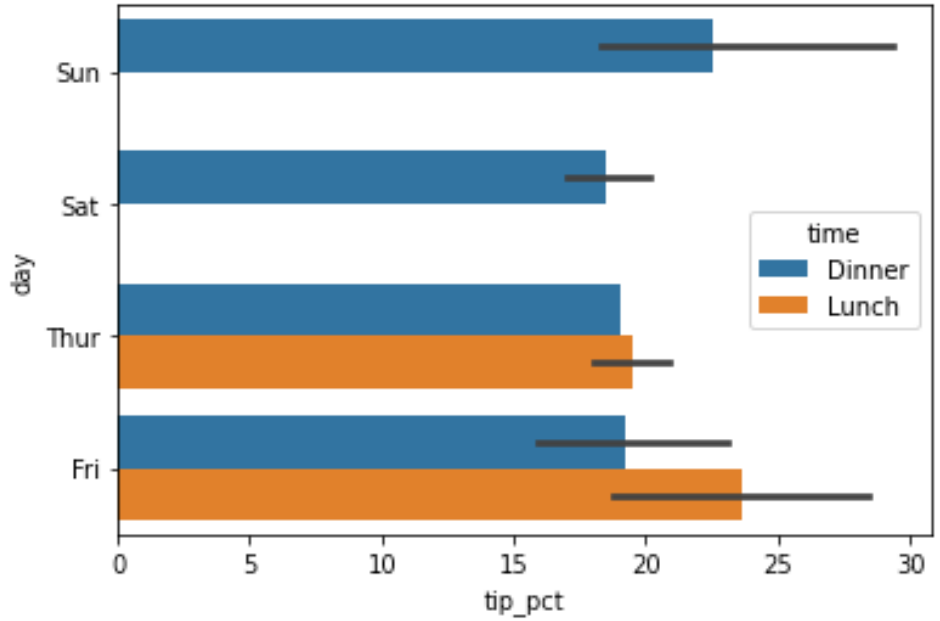
	total_bill	tip	smoker	day	time	size
0	16.99	1.01	No	Sun	Dinner	2
1	10.34	1.66	No	Sun	Dinner	3
2	21.01	3.50	No	Sun	Dinner	3
3	23.68	3.31	No	Sun	Dinner	2
4	24.59	3.61	No	Sun	Dinner	4
...	...	...	...	...	...	...
239	29.03	5.92	No	Sat	Dinner	3
240	27.18	2.00	Yes	Sat	Dinner	2
241	22.67	2.00	Yes	Sat	Dinner	2
242	17.82	1.75	No	Sat	Dinner	2
243	18.78	3.00	No	Thur	Dinner	2

244 rows × 6 columns

>> Bar plot with seaborn. Notice the error line?

```
tips['tip_pct'] = tips['tip']*100/(tips['total_bill'] - tips['tip'])
sns.barplot(data=tips, x='tip_pct', y='day', hue='time', orient='h')
```

```
<AxesSubplot:xlabel='tip_pct', ylabel='day'>
```



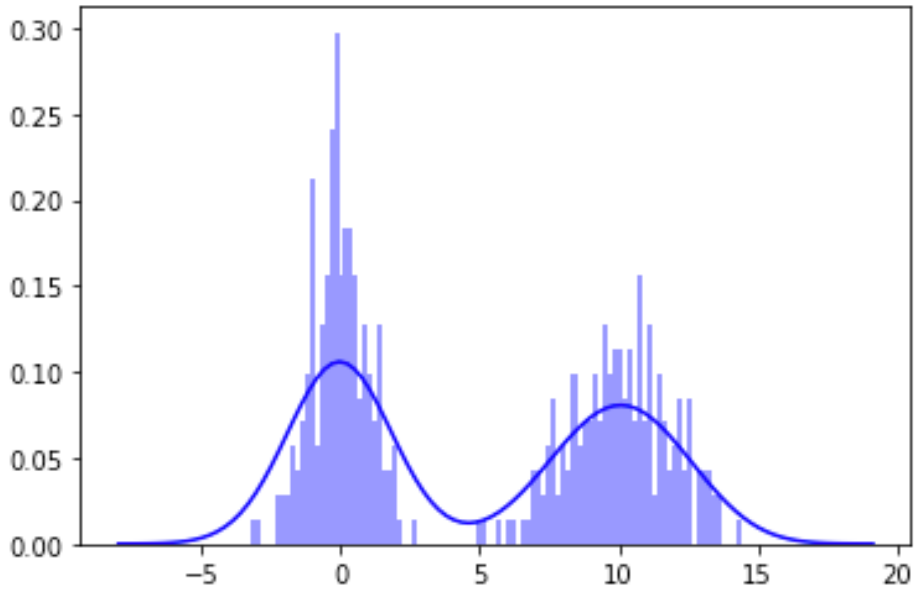
>> 4.2.2: Plotting with Pandas and Seaborn

>> Seaborn also did its own calculation under the hood and present data beautifully.

>> More than just histogram this is a distribution plot

```
fig, axs = plt.subplots(1, 1)
dist1 = np.random.normal(0, 1, size=200)
dist2 = np.random.normal(10, 2, size=200)
dist = pd.Series(np.concatenate([dist1, dist2]))
sns.distplot(dist, bins=100, color='b')
```

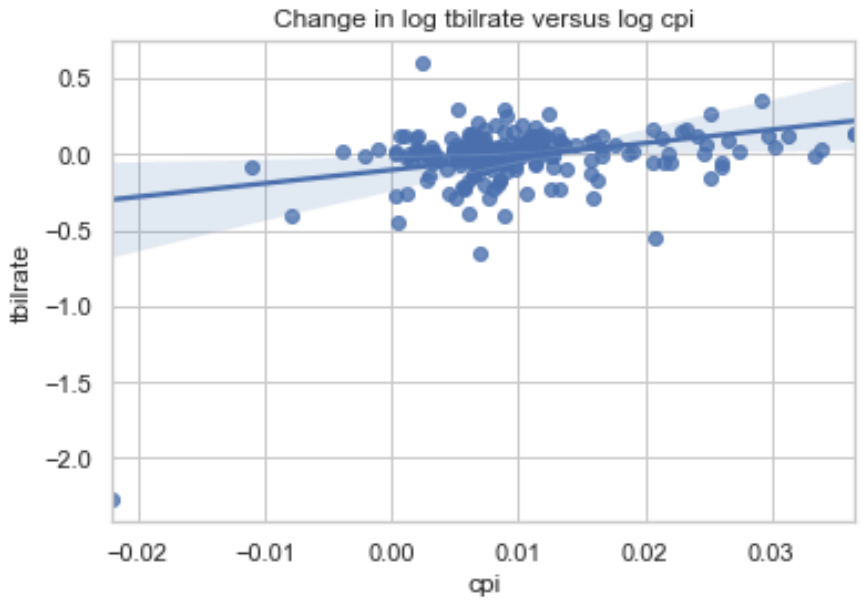
```
<AxesSubplot:>
```



>> Seaborn regression scatter plot

```
sns.set(style='whitegrid')
df = pd.read_csv('macrodata.csv')
data = df[['cpi', 'tbilrate', 'realdpi', 'realcons']]
trans_data = np.log(data).diff().dropna()
fig = sns.regplot(data=trans_data, x='cpi', y='tbilrate')
fig.set_title('Change in log {} versus log {}'.format('tbilrate', 'cpi'))
```

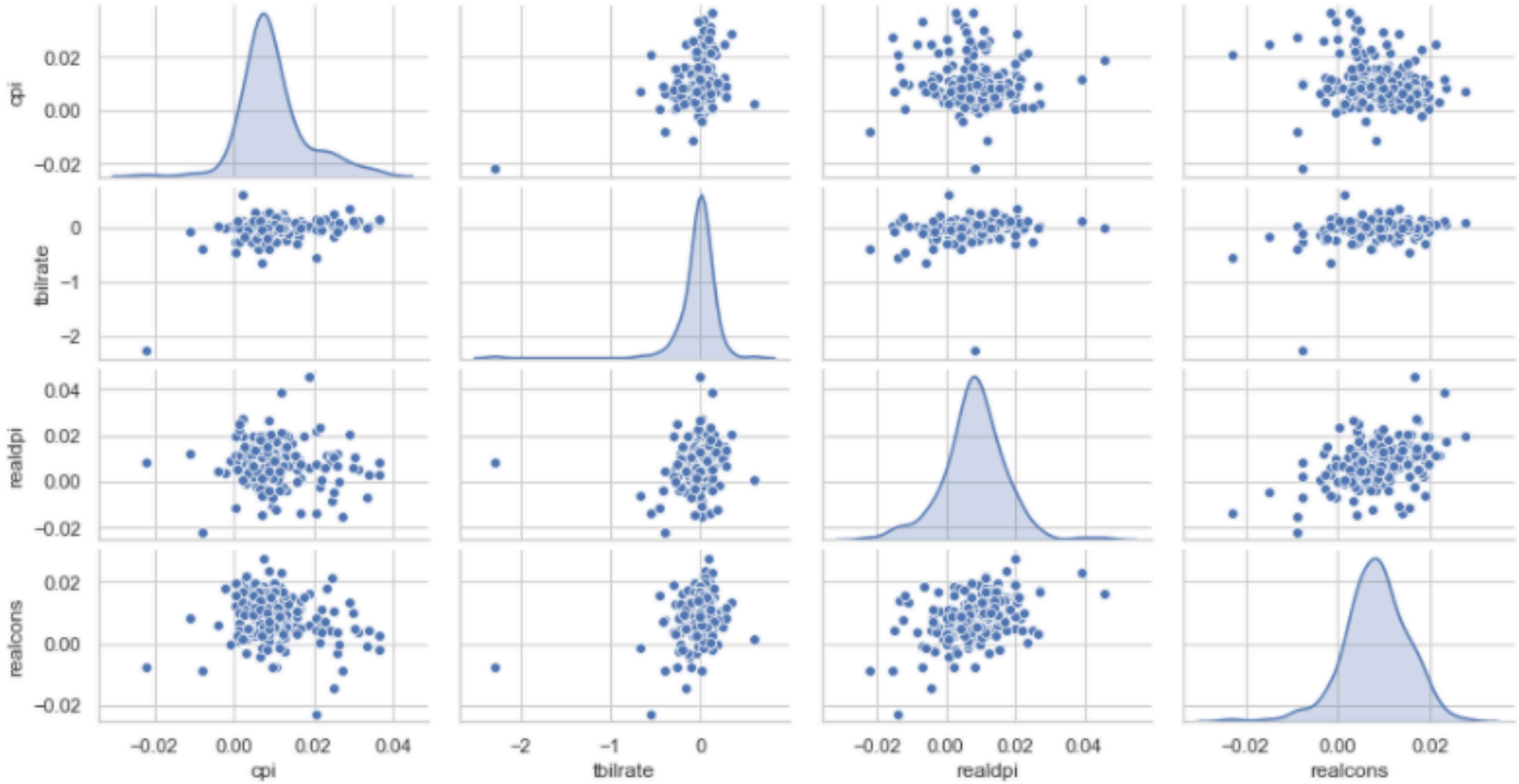
```
Text(0.5, 1.0, 'Change in log tbilrate versus log cpi')
```



>> 4.2.2: Plotting with Pandas and Seaborn

>> Seaborn pairplot will do pair-wise analysis and plotting for all columns of the dataset

```
grid = sns.pairplot(data=trans_data, diag_kind='kde')
grid.fig.set_size_inches(12,6)
```



```
>> PRINT("THAT'S FOR TODAY, FOLKS!!!")
>> EXIT()
```