Lecture 5:
Data Aggregation & Group Operations
Time Series Data

Section 5.1 Data Aggregation Group Operations

Example of relational database

5.1.1: Relational Databases

ENTRY

event_id	horse_id	place
0101	101	1
0101	102	2
0101	201	3
0101	301	4
0102	201	2
0103	102	3
0201	101	1
0301	301	2
0401	102	7
0501	102	1
0501	301	3

EVENT

event_id	show_id	event_name	judge_id
0101	01	Dressage	01
0102	01	Jumping	02
0103	01	Led in	01
0201	02	Led in	02
0301	03	Led in	01
0401	04	Dressage	04
0501	05	Dressage	01
0502	05	Flag and Pole	02

HORSE

horse_id	horse_name	colour	sire	dam	born	died	gender
101	Flash	white	201	301	1990	*NULL*	S
102	Star	brown	201	302	1991	*NULL*	M
201	Boxer	grey	401	501	1980	*NULL*	S
301	Daisy	white	401	502	1981	*NULL*	M
302	Tinkle	brown	401	501	1981	1994	M
401	Snowy	white	*NULL*	*NULL*	1976	1984	S
501	Bluebell	grey	*NULL*	*NULL*	1975	1982	M
502	Sally	white	*NULL*	*NULL*	1974	1987	M

PRIZE

event_code	place	prizemoney
0101	1	120
0101	2	60
0101	3	30
0102	1	10
0102	2	5
0103	1	100
0103	2	60
0103	3	40
0201	1	10
0201	2	5
0401	1	1000
0401	2	500
0401	3	250
0501	1	10
0501	2	5

JUDGE

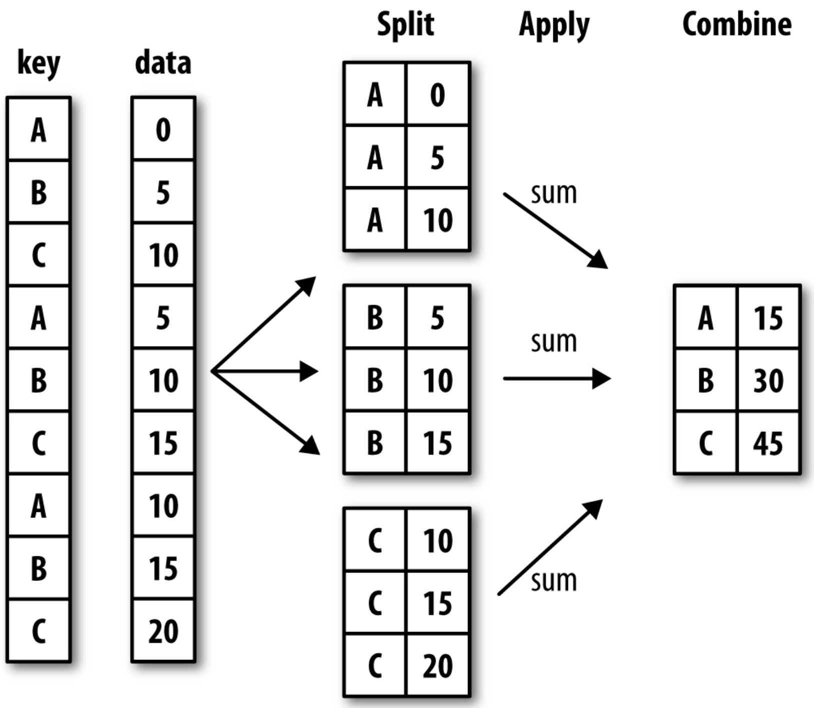
judge_id	judge_name	address
01	Smith	Melbourne
02	Green	Cootamundra
03	Gates	Dunkeld
04	Smith	Sydney

SHOW

show_id	show_name	show_held	show_address
01	Dubbo	1995-07-05 00:00:00	23 Wingewarra St, Dubbo
02	Young	1995-09-13 00:00:00	13 Cherry Lane, Young
03	Castle Hill	1996-05-04 00:00:00	Showground Rd, Castle Hill
04	Royal Easter	0000-00-00 00:00:00	PO Box 13, GPO Sydney
05	Dubbo	1996-07-01 00:00:00	17 Fitzroy St, Dubbo

5.1.1: Relational Databases

- Data is usually splited into tables, each with their own primary key(s)
- Merge (Join) is used to combine data from multiple tables
- GroupBy is used to aggregate data within the same table




```sql
SELECT User.Name, Category.Name, COUNT(Post.*)
  FROM Post
  JOIN User ON Post.AuthorID = User.UserID
  JOIN Category ON Category.CategoryID = Post.CategoryID
  GROUP BY User.UserID, Category.CategoryID
```

User Table

UserID	Name	Email	CreatedAt	UpdatedAt
1	sven	sven@your_app.com	2014-08-01 23:14:34	2014-08-01 23:14:34
2	hans	hans@another_app.com	2014-08-04 02:43:22	2014-08-04 02:43:22
3	olaf	olaf@super_app.com	2014-08-06 06:12:10	2014-08-06 06:12:10
4	beorn	beorn@app.com	2014-08-08 09:40:58	2014-08-08 09:40:58
5	smellyoaf	olaf@super_app.com	2014-08-10 13:09:46	2014-08-10 13:09:46
6	stig	beorn@app.com	2014-08-12 16:38:34	2014-08-12 16:38:34
7	siverth	olaf@super_app.com	2014-08-14 20:07:22	2014-08-14 20:07:22
8	gunilla	beorn@app.com	2014-08-16 23:36:10	2014-08-16 23:36:10

Foreign Keys to JOIN on

Post Table

PostID	Body	AuthorID	CategoryID	CreatedAt	UpdatedAt
1	The first post!	2	1	2014-08-01 23:14:34	2014-08-01 23:14:34
2	The second post!	1	1	2014-08-05 02:38:20	2014-08-05 02:38:20
3	The third post!	1	3	2014-08-08 06:02:05	2014-08-08 06:02:05
4	The fourth post!	3	3	2014-08-11 09:25:51	2014-08-11 09:25:51
5	The fifth post!	2	2	2014-08-14 12:49:36	2014-08-14 12:49:36
6	The sixth post!	2	3	2014-08-17 16:13:22	2014-08-17 16:13:22
7	The seventh post!	1	1	2014-08-20 19:37:08	2014-08-20 19:37:08
8	The eighth post!	3	3	2014-08-23 23:00:53	2014-08-23 23:00:53

Category Table

CategoryID	Name
1	funny
2	sad
3	geeky stuff

Results

User.Name	Category.Name	Count(*)
sven	funny	2
sven	geeky stuff	1
hans	funny	1
hans	sad	1
hans	geeky stuff	1
olaf	geeky stuff	2

5.1.2: GroupBy Mechanics

GroupBy Mechanics

	species	sepal_length	sepal_width	petal_length	petal_width
0	setosa	5.1	3.5	1.4	0.2
1	setosa	4.9	3.0	1.4	0.2
2	setosa	4.7	3.2	1.3	0.2
3	setosa	4.6	3.1	1.5	0.2
4	setosa	5.0	3.6	1.4	0.2
50	versicolor	7.0	3.2	4.7	1.4
51	versicolor	6.4	3.2	4.5	1.5
52	versicolor	6.9	3.1	4.9	1.5
53	versicolor	5.5	2.3	4.0	1.3
54	versicolor	6.5	2.8	4.6	1.5
100	virginica	6.3	3.3	6.0	2.5
101	virginica	5.8	2.7	5.1	1.9
102	virginica	7.1	3.0	5.9	2.1
103	virginica	6.3	2.9	5.6	1.8
104	virginica	6.5	3.0	5.8	2.2

SUM

SUM

SUM

species	sepal_length	sepal_width	petal_length	petal_width
setosa	24.3	16.4	7.0	1.0
versicolor	32.3	14.6	22.7	7.2
virginica	32.0	14.9	28.4	10.5

5.1.2: GroupBy Mechanics

Prepare a Dataframe

```
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
```

	key1	key2	data1	data2
0	a	one	0.948165	-0.156573
1	a	two	1.386119	1.661537
2	b	one	0.151955	-0.834981
3	b	two	-0.685776	-1.005415
4	a	one	0.311678	-1.681826

GroupBy with 2 keys

```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
```

```
key1  key2
a     one     0.629922
      two     1.386119
b     one     0.151955
      two    -0.685776
Name: data1, dtype: float64
```

```
means.unstack()
```

key2	one	two
key1		
a	0.629922	1.386119
b	0.151955	-0.685776

GroupBy with 1 key

```
grouped = df['data1'].groupby(df['key1'])
grouped.mean()
```

```
key1
a     0.881987
b    -0.266911
Name: data1, dtype: float64
```

MAGIC CODE INSTITUTE
Leverage your tech skills

5.1.2: GroupBy Mechanics

➢ GroupBy always goes with an aggregate function (sum(), mean(), size(), etc)

```
df.groupby(['key1', 'key2']).size()
```

```
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

```
df.groupby(['key1']).size()
```

```
key1
a    3
b    2
dtype: int64
```

```
df.groupby(['key2']).size()
```

```
key2
one    3
two    2
dtype: int64
```

```
df.groupby(['key1', 'key2']).sum()
```

key1	key2	data1	data2
a	one	1.259843	-1.838399
	two	1.386119	1.661537
b	one	0.151955	-0.834981
	two	-0.685776	-1.005415

```
df.groupby(['key1']).sum()
```

key1	data1	data2
a	2.645962	-0.176862
b	-0.533822	-1.840396

```
df.groupby(['key1', 'key2']).mean()
```

key1	key2	data1	data2
a	one	0.629922	-0.919199
	two	1.386119	1.661537
b	one	0.151955	-0.834981
	two	-0.685776	-1.005415

```
df.groupby(['key1']).mean()
```

key1	data1	data2
a	0.881987	-0.058954
b	-0.266911	-0.920198

5.1.2: GroupBy Mechanics

- For large datasets, it may be desirable to aggregate only a few columns.
- This is how you group by for specific columns

```
df.groupby(['key1', 'key2'])[['data2']].mean()
```

key1	key2	data2
a	one	-0.919199
	two	1.661537
b	one	-0.834981
	two	-1.005415

```
df.groupby(['key1', 'key2'])[['data1']].mean()
```

key1	key2	data1
a	one	0.629922
	two	1.386119
b	one	0.151955
	two	-0.685776

```
df.groupby(['key1'])[['data1']].mean()
```

key1	data1
a	0.881987
b	-0.266911

```
df.groupby(['key1'])[['data2']].mean()
```

key1	data2
a	-0.058954
b	-0.920198

```
df.groupby(['key2'])[['data1']].mean()
```

key2	data1
one	0.470599
two	0.350171

```
df.groupby(['key2'])[['data2']].mean()
```

key2	data2
one	-0.891127
two	0.328061

- Grouping with function columns

```python
people = pd.DataFrame(np.random.randn(5, 5),
         columns=['a', 'b', 'c', 'd', 'e'],
         index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
people
```

	a	b	c	d	e
Joe	-0.093569	0.656708	0.435519	-0.141911	0.623017
Steve	0.929007	1.049534	-0.314017	-1.441519	0.976015
Wes	-0.156138	1.082187	0.698172	-0.181940	-0.827780
Jim	-1.993911	0.467342	-1.827834	-1.087807	0.434433
Travis	1.395265	-1.661961	1.091080	0.392148	-1.145635

```python
people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values
people
```

	a	b	c	d	e
Joe	-0.093569	0.656708	0.435519	-0.141911	0.623017
Steve	0.929007	1.049534	-0.314017	-1.441519	0.976015
Wes	-0.156138	NaN	NaN	-0.181940	-0.827780
Jim	-1.993911	0.467342	-1.827834	-1.087807	0.434433
Travis	1.395265	-1.661961	1.091080	0.392148	-1.145635

```python
people.groupby(len).sum()
```

	a	b	c	d	e
3	-2.243617	1.124050	-1.392314	-1.411658	0.229670
5	0.929007	1.049534	-0.314017	-1.441519	0.976015
6	1.395265	-1.661961	1.091080	0.392148	-1.145635

- Supported aggregate functions:

Function name	Description
`count`	Number of non-NA values in the group
`sum`	Sum of non-NA values
`mean`	Mean of non-NA values
`median`	Arithmetic median of non-NA values
`std, var`	Unbiased (n – 1 denominator) standard deviation and variance
`min, max`	Minimum and maximum of non-NA values
`prod`	Product of non-NA values

- Define your own aggregate function:

```
def peak_to_peak(arr):
    return arr.max() - arr.min()

grouped = df.groupby('key1')
grouped.agg(peak_to_peak)
```

key1	data1	data2
a	1.074441	3.343363
b	0.837731	0.170434

- methods like describe() also work, even though they are not aggregations:

```
grouped['data1'].describe()
```

key1	count	mean	std	min	25%	50%	75%	max
a	3.0	0.881987	0.540269	0.311678	0.629922	0.948165	1.167142	1.386119
b	2.0	-0.266911	0.592365	-0.685776	-0.476344	-0.266911	-0.057478	0.151955

```
grouped['data2'].describe()
```

key1	count	mean	std	min	25%	50%	75%	max
a	3.0	-0.058954	1.673818	-1.681826	-0.919199	-0.156573	0.752482	1.661537
b	2.0	-0.920198	0.120515	-1.005415	-0.962807	-0.920198	-0.877590	-0.834981

➢ Column-Wise and Multiple Function Application

```python
tips = pd.read_csv('tips.csv')
tips['tip_pct'] = tips['tip'] / tips['total_bill']
tips
```

	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
...	...	...	...	...	...	...	...
239	29.03	5.92	No	Sat	Dinner	3	0.203927
240	27.18	2.00	Yes	Sat	Dinner	2	0.073584
241	22.67	2.00	Yes	Sat	Dinner	2	0.088222
242	17.82	1.75	No	Sat	Dinner	2	0.098204
243	18.78	3.00	No	Thur	Dinner	2	0.159744

244 rows × 7 columns

```python
grouped = tips.groupby(['day', 'smoker'])
grouped_pct = grouped['tip_pct']
grouped_pct.agg('mean')
```

```
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
```

```python
grouped_pct.agg(['mean', 'std', peak_to_peak])
```

day	smoker	mean	std	peak_to_peak
Fri	**No**	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	**No**	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	**No**	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	**No**	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

- you can specify a list of functions to apply to all of the columns or different functions per column
- suppose we wanted to compute the same

```
functions = ['count', 'mean', 'max']
result = grouped['tip_pct', 'total_bill'].agg(functions)
result
```

		tip_pct			total_bill		
		count	mean	max	count	mean	max
day	smoker						
Fri	No	4	0.151650	0.187735	4	18.420000	22.75
	Yes	15	0.174783	0.263480	15	16.813333	40.17
Sat	No	45	0.158048	0.291990	45	19.661778	48.33
	Yes	42	0.147906	0.325733	42	21.276667	50.81
Sun	No	57	0.160113	0.252672	57	20.506667	48.17
	Yes	19	0.187250	0.710345	19	24.120000	45.35
Thur	No	45	0.160298	0.266312	45	17.113111	41.19
	Yes	17	0.163863	0.241255	17	19.190588	43.11

- suppose you wanted to apply potentially different functions to one or more of the columns
- pass a dict to agg that contains a mapping of

```
grouped.agg({'tip' : np.max, 'size' : 'sum'})
```

		tip	size
day	smoker		
Fri	No	3.50	9
	Yes	4.73	31
Sat	No	9.00	115
	Yes	10.00	104
Sun	No	6.00	167
	Yes	6.50	49
Thur	No	6.70	112
	Yes	5.00	40

5.1.3: Data Aggregation

➢ Or different set of multiple formulas for each columns

```
grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
             'size' : ['count', 'sum', 'var', 'median']})
```

		tip_pct				size			
		min	max	mean	std	count	sum	var	median
day	smoker								
Fri	No	0.120385	0.187735	0.151650	0.028123	4	9	0.250000	2
	Yes	0.103555	0.263480	0.174783	0.051293	15	31	0.352381	2
Sat	No	0.056797	0.291990	0.158048	0.039767	45	115	0.616162	2
	Yes	0.035638	0.325733	0.147906	0.061375	42	104	0.743322	2
Sun	No	0.059447	0.252672	0.160113	0.042347	57	167	1.066416	3
	Yes	0.065660	0.710345	0.187250	0.154134	19	49	0.812865	2
Thur	No	0.072961	0.266312	0.160298	0.038774	45	112	1.391919	2
	Yes	0.090014	0.241255	0.163863	0.039389	17	40	0.492647	2

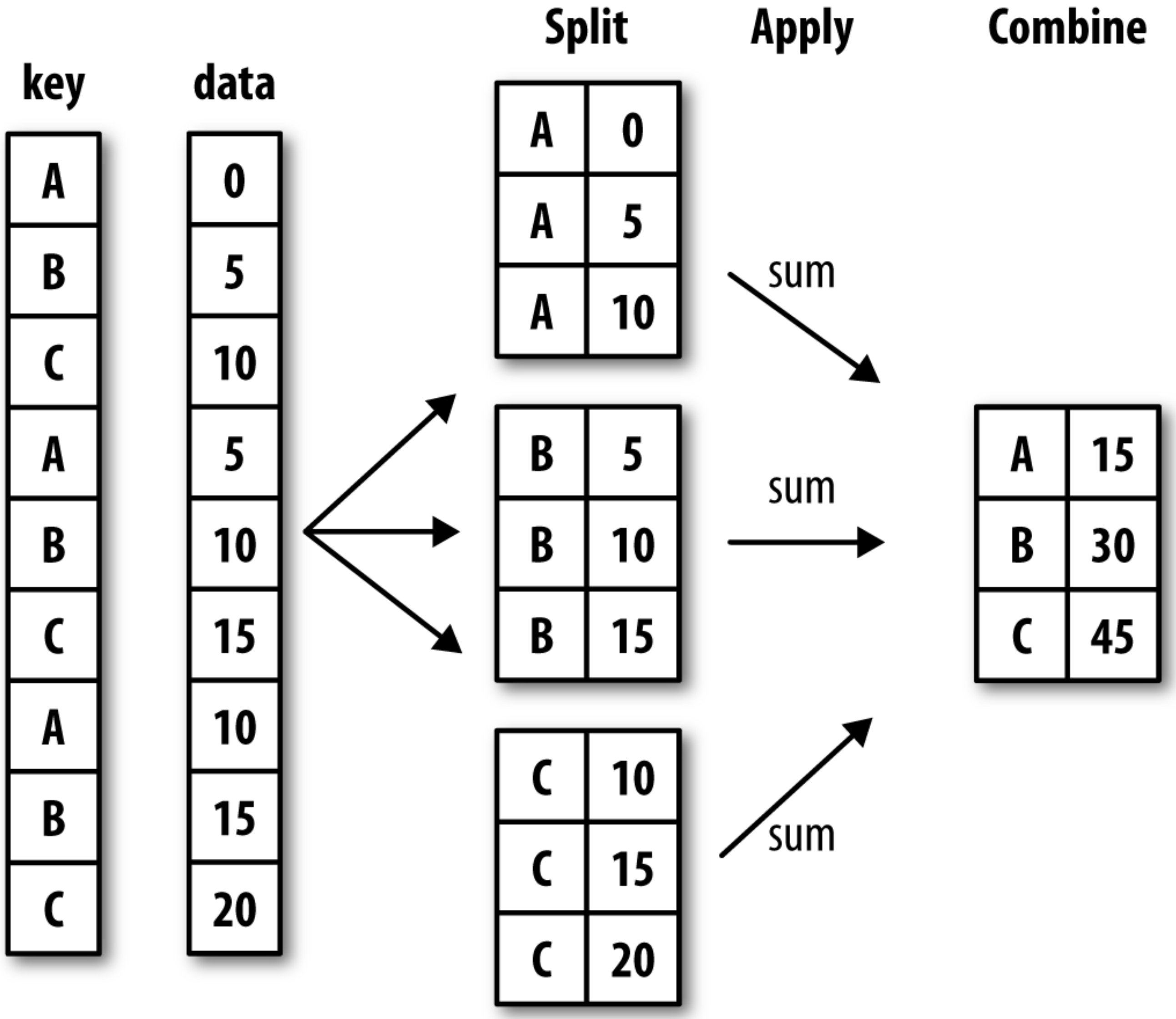
➢ Returning Aggregated Data Without (Hierarchical) Row Indexes

```
tips.groupby(['day', 'smoker'], as_index=False).mean()
```

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

5.1.4: Apply : split – apply - combine

- `apply` splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concatenate the pieces together.
- `apply` is the most general-purpose GroupBy method



5.1.4: Apply : split – apply - combine

- Suppose you wanted to select the top five tip_pct values by group.
- First, write a function that selects the rows with the largest values in a particular column:

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

top(tips, n=5)
```

	total_bill	tip	smoker	day	time	size	tip_pct
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

- Group By "smoker", and call apply with this function

```
tips.groupby('smoker').apply(top)
```

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
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

- The `top` function is called on each row group from the DataFrame.
- Then the results are glued together using `pandas.concat`, labeling the pieces with the group names.
- The result therefore has a hierarchical index whose inner level contains index values from the original DataFrame.

5.1.4: Apply : split – apply - combine

- If you pass a function to apply that takes other arguments or keywords, you can pass these after the function:

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

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

- disable hierarchical index by passing group_keys=False to groupby

```
tips.groupby('smoker', group_keys=False).apply(top)
```

	total_bill	tip	smoker	day	time	size	tip_pct
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
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

Filling missing values with Group-Specific Values with GroupBy & Apply

5.1.5: Example

- Create data with NaN values:

```
s = pd.Series(np.random.randn(6))
s[::2] = np.nan
s
```

```
0         NaN
1    0.849685
2         NaN
3   -1.107675
4         NaN
5   -1.204737
dtype: float64
```

- Fill NaN values with mean:

```
s.fillna(s.mean())
```

```
0   -0.487576
1    0.849685
2   -0.487576
3   -1.107675
4   -0.487576
5   -1.204737
dtype: float64
```

```
states = ['Ohio', 'New York', 'Vermont', 'Florida',
          'Oregon', 'Nevada', 'California', 'Idaho']
group_key = ['East'] * 4 + ['West'] * 4
data = pd.Series(np.random.randn(8), index=states)
data
```

```
Ohio          1.607384
New York     -0.797976
Vermont       0.324115
Florida       0.944165
Oregon        1.335250
Nevada       -1.293500
California    0.133856
Idaho         0.233619
dtype: float64
```

```
data[['Vermont', 'Nevada', 'Idaho']] = np.nan
data
```

```
Ohio          1.607384
New York     -0.797976
Vermont            NaN
Florida       0.944165
Oregon        1.335250
Nevada             NaN
California    0.133856
Idaho              NaN
dtype: float64
```

- Fill values by group

```
data.groupby(group_key).mean()
```

```
East    0.584524
West    0.734553
dtype: float64
```

```
fill_mean = lambda g: g.fillna(g.mean())
data.groupby(group_key).apply(fill_mean)
```

```
Ohio          1.607384
New York     -0.797976
Vermont       0.584524
Florida       0.944165
Oregon        1.335250
Nevada        0.734553
California    0.133856
Idaho         0.734553
dtype: float64
```

5.1.6: Pivot Tables

- Pivot tables in Python with pandas combines the groupby facility with reshape operations utilizing hierarchical indexing.
- The following 2 statements produce exactly the same result

```
tips.groupby(['day','smoker']).mean()
```

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

Default param: `aggfunc=np.mean`

```
tips.pivot_table(index=['day', 'smoker'])
```

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

- suppose we want to:
  - aggregate only `tip_pct` and `size`, and additionally group by `time`.
  - put `smoker` in the table columns and `day` in the rows:

```
tips.pivot_table(['tip_pct', 'size'],
                 index=['time', 'day'],
                 columns='smoker')
```

		size		tip_pct	
	smoker	No	Yes	No	Yes
time	day				
Dinner	Fri	2.000000	2.222222	0.139622	0.165347
	Sat	2.555556	2.476190	0.158048	0.147906
	Sun	2.929825	2.578947	0.160113	0.187250
	Thur	2.000000	NaN	0.159744	NaN
Lunch	Fri	3.000000	1.833333	0.187735	0.188937
	Thur	2.500000	2.352941	0.160311	0.163863

- augment this table to include partial totals by passing `margins=True`:
  - adding All row and column labels, with corresponding values being the group statistics for all the data within a single tier

```
tips.pivot_table(['tip_pct', 'size'],
                 index=['time', 'day'],
                 columns='smoker', margins=True)
```

		size			tip_pct		
	smoker	No	Yes	All	No	Yes	All
time	day						
Dinner	Fri	2.000000	2.222222	2.166667	0.139622	0.165347	0.158916
	Sat	2.555556	2.476190	2.517241	0.158048	0.147906	0.153152
	Sun	2.929825	2.578947	2.842105	0.160113	0.187250	0.166897
	Thur	2.000000	NaN	2.000000	0.159744	NaN	0.159744
Lunch	Fri	3.000000	1.833333	2.000000	0.187735	0.188937	0.188765
	Thur	2.500000	2.352941	2.459016	0.160311	0.163863	0.161301
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

5.1.6: Pivot Tables

- To use a different aggregate function:

```
tips.pivot_table('tip_pct', index=['time', 'smoker'],
                 columns='day', aggfunc=len, margins=True)
```

time	smoker	Fri	Sat	Sun	Thur	All
Dinner	No	3.0	45.0	57.0	1.0	106.0
	Yes	9.0	42.0	19.0	NaN	70.0
Lunch	No	1.0	NaN	NaN	44.0	45.0
	Yes	6.0	NaN	NaN	17.0	23.0
All		19.0	87.0	76.0	62.0	244.0

```
tips.pivot_table('tip_pct', index=['time', 'smoker'],
                 columns='day', aggfunc=np.sum, margins=True)
```

time	smoker	Fri	Sat	Sun	Thur	All
Dinner	No	0.418867	7.112145	9.126438	0.159744	16.817194
	Yes	1.488126	6.212055	3.557756	NaN	11.257937
Lunch	No	0.187735	NaN	NaN	7.053669	7.241404
	Yes	1.133620	NaN	NaN	2.785676	3.919295
All		3.228348	13.324199	12.684194	9.999089	39.235830

```
tips.pivot_table('tip_pct', index=['time', 'smoker'],
                 columns='day', aggfunc=np.size, margins=True)
```

time	smoker	Fri	Sat	Sun	Thur	All
Dinner	No	3.0	45.0	57.0	1.0	106.0
	Yes	9.0	42.0	19.0	NaN	70.0
Lunch	No	1.0	NaN	NaN	44.0	45.0
	Yes	6.0	NaN	NaN	17.0	23.0
All		19.0	87.0	76.0	62.0	244.0

```
tips.pivot_table('tip_pct', index=['time', 'smoker'],
                 columns='day', aggfunc=np.std, margins=True)
```

time	smoker	Fri	Sat	Sun	Thur	All
Dinner	No	0.017841	0.039767	0.042347	NaN	0.040458
	Yes	0.052676	0.061375	0.154134	NaN	0.095153
Lunch	No	NaN	NaN	NaN	0.039222	0.038989
	Yes	0.050262	NaN	NaN	0.039389	0.042770
All		0.047665	0.051293	0.084739	0.038652	0.060947

Section 5.2 | Time Series Data

5.2.1: Time Series Data - Overview

- Time Series: Anything that is observed or measured at many points in time.
  - Fixed frequency: data points occur at regular intervals according to some rule
    - every 15 seconds,
    - every 5 minutes,
    - once per month.
  - Irregular:
    - without a fixed unit of time or offset between units
- Time Series Data Types:
  - ***Timestamps***: specific instants in time
  - Fixed ***periods***: such as the month January 2020 or the full year 2019
  - ***Intervals*** of time: indicated by a start and end timestamp.

    Periods can be thought of as special cases of intervals

- datetime: stores both the date and time down to the microsecond

```
now = datetime.now()
now
```

```
datetime.datetime(2020, 3, 29, 12, 18, 15, 132932)
```

```
now.year, now.month, now.day
```

```
(2020, 3, 29)
```

- timedelta: represents the temporal difference between two datetime objects

```
delta = datetime(2029, 3, 30) - datetime(2019, 12, 31, 8, 15)
delta
```

```
datetime.timedelta(days=3376, seconds=56700)
```

```
delta.days, delta.seconds
```

```
(3376, 56700)
```

- datetime: Add or Subtract timedelta

```
start = datetime(2020, 3, 30)
start + timedelta(31)
```

```
datetime.datetime(2020, 4, 30, 0, 0)
```

```
start - 2 * timedelta(15)
```

```
datetime.datetime(2020, 2, 29, 0, 0)
```

➢ `datetime`: types supported

Type	Description
`date`	Store calendar date (year, month, day) using the Gregorian calendar
`time`	Store time of day as hours, minutes, seconds, and microseconds
`datetime`	Stores both date and time
`timedelta`	Represents the difference between two `datetime` values (as days, seconds, and microseconds)
`tzinfo`	Base type for storing time zone information

➢ Converting between string & datetime using strftime & strptime

```python
stamp = datetime(2011, 1, 3)
str(stamp)
```

```
'2011-01-03 00:00:00'
```

```python
stamp.strftime('%Y-%m-%d')
```

```
'2011-01-03'
```

```python
value = '2011-01-03'
datetime.strptime(value, '%Y-%m-%d')
```

```
datetime.datetime(2011, 1, 3, 0, 0)
```

```python
datestrs = ['7/6/2011', '8/6/2011']
[datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
```

```
[datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]
```

➢ Using dateutil.parser.parse: no format needed

```python
from dateutil.parser import parse
parse('2020-01-03')
```

```
datetime.datetime(2020, 1, 3, 0, 0)
```

```python
parse('Jan 31, 2020 10:45 PM')
```

```
datetime.datetime(2020, 1, 31, 22, 45)
```

```python
#if date appears before month
parse('6/12/2020', dayfirst=True)
```

```
datetime.datetime(2020, 12, 6, 0, 0)
```

➢ `To_datetime` method:

```
datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']
pd.to_datetime(datestrs)
```

```
DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='datetime64[ns]', freq=None)
```

```
#It also handles values that should be considered missing (None, empty string, etc.)
idx = pd.to_datetime(datestrs + [None])
idx
```

```
DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dtype='datetime64[ns]', freq=None)
```

```
idx[2]
```

```
NaT
```

```
pd.isnull(idx)
```

```
array([False, False,  True])
```

- Create a `pd.Series` with `datetime` index:

```
dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
         datetime(2011, 1, 7), datetime(2011, 1, 8),
         datetime(2011, 1, 10), datetime(2011, 1, 12)]

ts = pd.Series(np.random.randn(6), index=dates)
ts
```

```
2011-01-02   -1.216694
2011-01-05    0.593616
2011-01-07   -1.126609
2011-01-08   -0.205146
2011-01-10    0.875307
2011-01-12   -0.184089
dtype: float64
```

```
ts.index
```

```
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

- Indexing, Selection,

```
ts.index[0]
```

```
Timestamp('2011-01-02 00:00:00')
```

```
stamp = ts.index[2]
ts[stamp]
```

```
-1.1266085369311785
```

```
ts['1/10/2011']
```

```
0.8753068499686161
```

```
ts['20110110']
```

```
0.8753068499686161
```

5.2.3: Time Series Basics with Pandas

- For longer time series:

```
longer_ts = pd.Series(np.random.randn(1000),index=pd.date_range('1/1/2000', periods=1000))
longer_ts
```

```
2000-01-01   -0.369458
2000-01-02   -0.565862
2000-01-03    2.078277
2000-01-04   -0.020037
2000-01-05    0.623318
                ...
2002-09-22    1.467914
2002-09-23   -1.412659
2002-09-24   -1.102563
2002-09-25    0.169510
2002-09-26    0.742637
Freq: D, Length: 1000, dtype: float64
```

- a year or only a year and month can be passed to easily select slices of data

```
longer_ts['2001']
```

```
2001-01-01    0.989035
2001-01-02    1.374789
2001-01-03    0.447148
2001-01-04    1.031572
2001-01-05   -0.018959
                ...
2001-12-27    2.448997
2001-12-28    0.135101
2001-12-29   -0.817247
2001-12-30    0.602396
2001-12-31   -0.203769
Freq: D, Length: 365, dtype: float64
```

```
longer_ts['2001-01']
```

```
2001-01-14    0.920879
2001-01-15   -0.071784
2001-01-16   -1.178991
2001-01-17    0.231834
2001-01-18   -1.254995
2001-01-19   -1.205983
2001-01-20   -0.983882
2001-01-21   -0.070365
2001-01-22   -0.166876
2001-01-23   -0.359498
2001-01-24    1.046763
2001-01-25   -0.349220
2001-01-26   -1.980147
2001-01-27   -1.641288
2001-01-28    0.048715
2001-01-29   -0.381381
2001-01-30    1.877826
2001-01-31    0.364842
Freq: D, dtype: float64
```

SCRIPT
MAGIC CODE INSTITUTE
Leverage your tech skills

- Generating Date Range:

```
index = pd.date_range('2012-04-01', '2012-06-01')
index
```

```
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

```
pd.date_range(start='2012-04-01', periods=20)
```

```
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')
```

5.2.4: Date Ranges, Frequencies, Shifting

- Date Range with start, end and frequency:

```
pd.date_range('2000-01-01', '2000-12-01', freq='BM')
```

```
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

- date_range by default preserves the time (if any) of the start or end timestamp:

```
pd.date_range('2012-05-02 12:56:31', periods=5)
```

```
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

- Putting an integer before the base frequency creates a multiple:

```
pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')
```

```
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

5.2.4: Date Ranges, Frequencies, Shifting

- Shifting (Leading and Lagging) Data

```
ts = pd.Series(np.random.randn(4),
               index=pd.date_range('1/1/2000',
                                   periods=4,
                                   freq='M'))
ts
```

```
2000-01-31    0.360993
2000-02-29    0.538253
2000-03-31    0.691856
2000-04-30   -0.518085
Freq: M, dtype: float64
```

```
ts.shift(1)
```

```
2000-01-31         NaN
2000-02-29    0.360993
2000-03-31    0.538253
2000-04-30    0.691856
Freq: M, dtype: float64
```

```
ts.shift(-1)
```

```
2000-01-31    0.538253
2000-02-29    0.691856
2000-03-31   -0.518085
2000-04-30         NaN
Freq: M, dtype: float64
```

```
#computing percent changes
ts / ts.shift(1) - 1
```

```
2000-01-31         NaN
2000-02-29    0.491035
2000-03-31    0.285372
2000-04-30   -1.748834
Freq: M, dtype: float64
```

- Shift both Timestamp & Data

```
ts.shift(2, freq='M')
```

```
2000-03-31    0.360993
2000-04-30    0.538253
2000-05-31    0.691856
2000-06-30   -0.518085
Freq: M, dtype: float64
```

```
ts.shift(3, freq='D')
```

```
2000-02-03    0.360993
2000-03-03    0.538253
2000-04-03    0.691856
2000-05-03   -0.518085
dtype: float64
```

```
ts.shift(1, freq='90T')
```

```
2000-01-31 01:30:00    0.360993
2000-02-29 01:30:00    0.538253
2000-03-31 01:30:00    0.691856
2000-04-30 01:30:00   -0.518085
Freq: M, dtype: float64
```

The “T” here stands for minutes.

5.2.5: Periods and Period Arithmetic

- Periods represent timespans, like days, months, quarters, or years.
- The Period class represents this data type, requiring a string or integer and a frequency

```
p = pd.Period(2007, freq='A-DEC')
p
```

```
Period('2007', 'A-DEC')
```

- adding and subtracting integers from periods has the effect of shifting by their frequency

```
p + 5
```

```
Period('2012', 'A-DEC')
```

```
p - 2
```

```
Period('2005', 'A-DEC')
```

- If two periods have the same frequency, their difference is the number of units between them

```
pd.Period('2014', freq='A-DEC') - p
```

```
<7 * YearEnds: month=12>
```

- ranges of periods can be constructed with the period_range function

```
rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')
rng
```

```
PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04',
             '2000-05', '2000-06'], dtype='period[M]', freq='M')
```

- The PeriodIndex class stores a sequence of periods and can serve as an axis index in any pandas data structure

```
pd.Series(np.random.randn(6), index=rng)
```

```
2000-01   -1.717816
2000-02    1.615029
2000-03   -1.392915
2000-04    0.378783
2000-05    1.072036
2000-06   -2.309436
Freq: M, dtype: float64
```

5.2.5: Periods and Period Arithmetic

Period Frequency Conversion

```
p = pd.Period('2007', freq='A-DEC')
p
```

```
Period('2007', 'A-DEC')
```

```
p.asfreq('M', how='start')
```

```
Period('2007-01', 'M')
```

```
p.asfreq('M', how='end')
```

```
Period('2007-12', 'M')
```

```
p = pd.Period('2007', freq='A-JUN')
p
```

```
Period('2007', 'A-JUN')
```

```
p.asfreq('M', 'start')
```

```
Period('2006-07', 'M')
```

```
p.asfreq('M', 'end')
```

```
Period('2007-06', 'M')
```

Quarterly Period Frequencies

```
p = pd.Period('2012Q4', freq='Q-JAN')
p
```

```
Period('2012Q4', 'Q-JAN')
```

```
p.asfreq('D', 'start')
```

```
Period('2011-11-01', 'D')
```

```
p.asfreq('D', 'end')
```

```
Period('2012-01-31', 'D')
```

generate quarterly ranges

```
rng = pd.period_range('2011Q3',
                      '2012Q4',
                      freq='Q-JAN')
ts = pd.Series(np.arange(len(rng)),
               index=rng)
ts
```

```
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int32
```

Converting Timestamps to Periods (and Back)

```
rng = pd.date_range('2000-01-01', periods=3, freq='M')
ts = pd.Series(np.random.randn(3), index=rng)
ts
```

```
2000-01-31   -0.956830
2000-02-29    1.319765
2000-03-31    0.114185
Freq: M, dtype: float64
```

```
pts = ts.to_period()
pts
```

```
2000-01   -0.956830
2000-02    1.319765
2000-03    0.114185
Freq: M, dtype: float64
```

- Resampling = converting a time series from one frequency to another.
  - downsampling = aggregating higher frequency data to lower frequency
  - upsampling = converting lower frequency to higher frequency is called.

```
rng = pd.date_range('2000-01-01', periods=100, freq='D')
ts = pd.Series(np.random.randn(len(rng)), index=rng)
ts
```

```
2000-01-01   -0.701248
2000-01-02   -0.548084
2000-01-03   -0.151535
2000-01-04    1.454100
2000-01-05    1.050801
                 ...
2000-04-05   -1.110327
2000-04-06    0.922107
2000-04-07    0.784788
2000-04-08   -0.091624
2000-04-09   -0.176648
Freq: D, Length: 100, dtype: float64
```

```
ts.resample('M').mean()
```

```
2000-01-31   -0.235264
2000-02-29   -0.198937
2000-03-31   -0.106851
2000-04-30   -0.020139
Freq: M, dtype: float64
```

```
ts.resample('M', kind='period').mean()
```

```
2000-01   -0.235264
2000-02   -0.198937
2000-03   -0.106851
2000-04   -0.020139
Freq: M, dtype: float64
```

5.2.6: Resampling & Frequency Conversion

Downsampling

```
rng = pd.date_range('2000-01-01', periods=12, freq='T')
ts = pd.Series(np.arange(12), index=rng)
ts
```

```
2000-01-01 00:00:00     0
2000-01-01 00:01:00     1
2000-01-01 00:02:00     2
2000-01-01 00:03:00     3
2000-01-01 00:04:00     4
2000-01-01 00:05:00     5
2000-01-01 00:06:00     6
2000-01-01 00:07:00     7
2000-01-01 00:08:00     8
2000-01-01 00:09:00     9
2000-01-01 00:10:00    10
2000-01-01 00:11:00    11
Freq: T, dtype: int32
```

```
ts.resample('5min', closed='right').sum()
```

```
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int32
```

Upsampling

```
frame = pd.DataFrame(np.random.randn(2, 4),
        index=pd.date_range('1/1/2000', periods=2,freq='W-WED'),
        columns=['Colorado', 'Texas', 'New York', 'Ohio'])
frame
```

	Colorado	Texas	New York	Ohio
2000-01-05	1.152088	0.285833	0.569334	-0.205589
2000-01-12	-1.264938	-1.547976	-0.756922	0.319351

```
df_daily = frame.resample('D').asfreq()
df_daily
```

	Colorado	Texas	New York	Ohio
2000-01-05	1.152088	0.285833	0.569334	-0.205589
2000-01-06	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	-1.264938	-1.547976	-0.756922	0.319351

- Fill forward

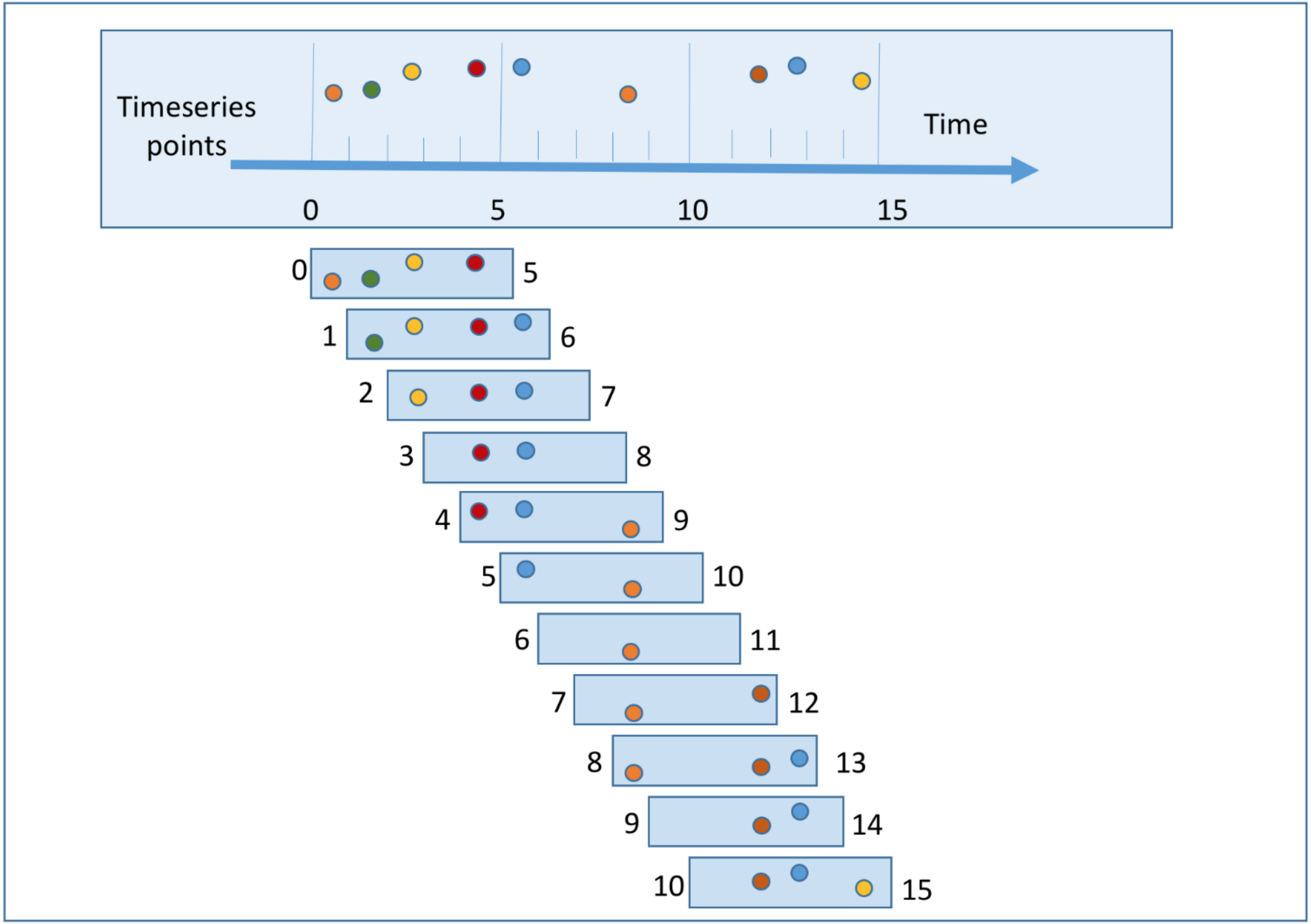
```
frame.resample('D').ffill()
```

	Colorado	Texas	New York	Ohio
2000-01-05	1.152088	0.285833	0.569334	-0.205589
2000-01-06	1.152088	0.285833	0.569334	-0.205589
2000-01-07	1.152088	0.285833	0.569334	-0.205589
2000-01-08	1.152088	0.285833	0.569334	-0.205589
2000-01-09	1.152088	0.285833	0.569334	-0.205589
2000-01-10	1.152088	0.285833	0.569334	-0.205589
2000-01-11	1.152088	0.285833	0.569334	-0.205589
2000-01-12	-1.264938	-1.547976	-0.756922	0.319351

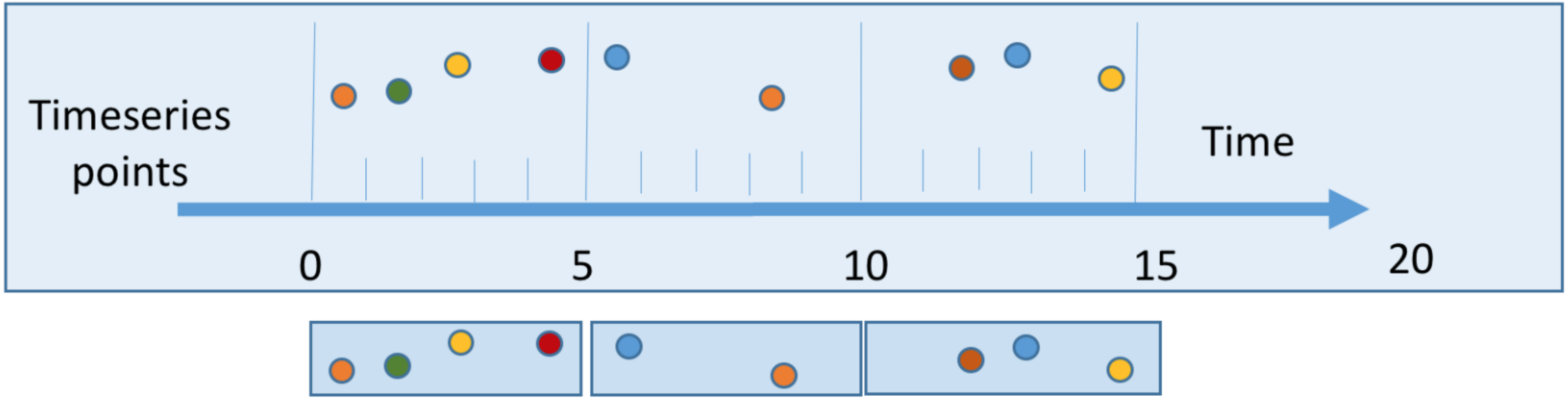
```
frame.resample('D').ffill(limit=2)
```

	Colorado	Texas	New York	Ohio
2000-01-05	1.152088	0.285833	0.569334	-0.205589
2000-01-06	1.152088	0.285833	0.569334	-0.205589
2000-01-07	1.152088	0.285833	0.569334	-0.205589
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	-1.264938	-1.547976	-0.756922	0.319351

5.2.7: Moving window functions

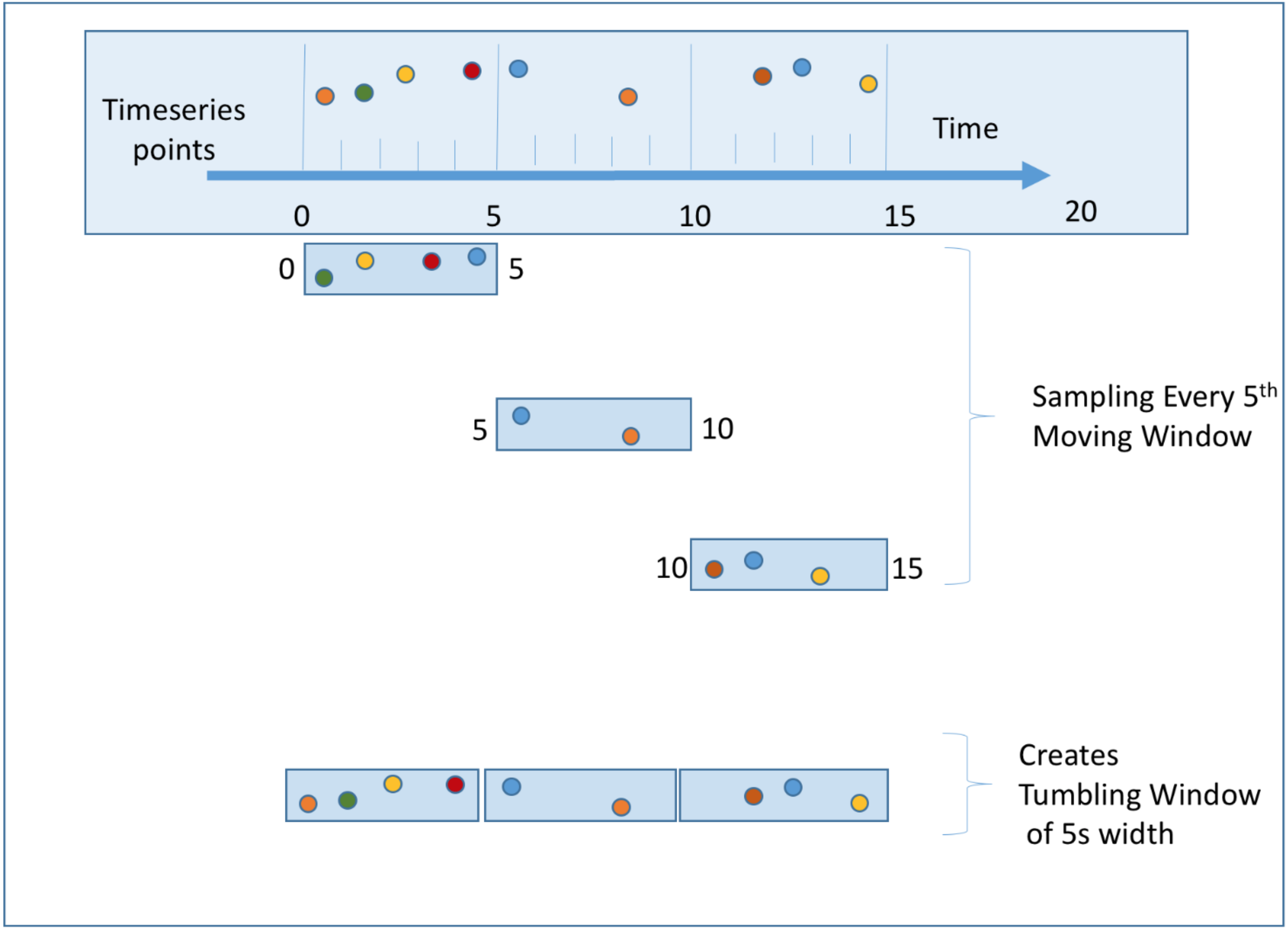


5.2.7: Moving window functions



MAGIC CODE INSTITUTE
Leverage your tech skills

5.2.7: Moving window functions



5.2.7: Moving window functions

- Useful for smoothing noisy data with:
  - Sliding window
  - Exponentially decaying weights

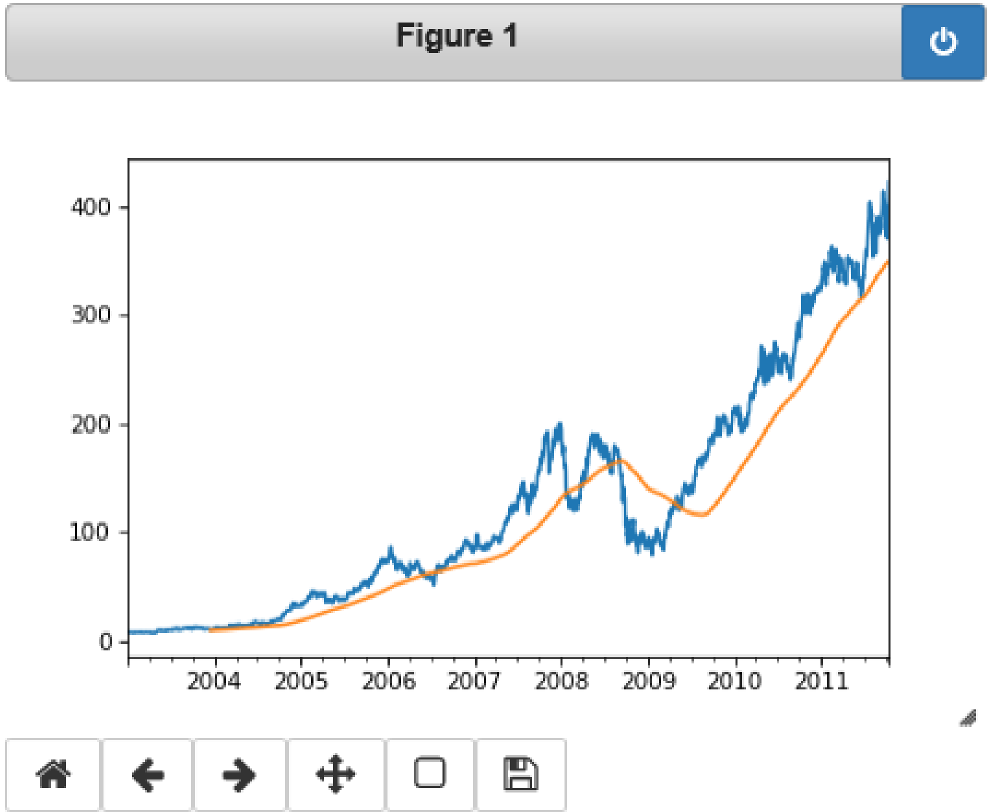
```
close_px_all = pd.read_csv('stock_px_2.csv',
                           parse_dates=True,
                           index_col=0)
close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]
close_px = close_px.resample('B').ffill()
close_px
```

	AAPL	MSFT	XOM
2003-01-02	7.40	21.11	29.22
2003-01-03	7.45	21.14	29.24
2003-01-06	7.45	21.52	29.96
2003-01-07	7.43	21.93	28.95
2003-01-08	7.28	21.31	28.83
...	...	...	...
2011-10-10	388.81	26.94	76.28
2011-10-11	400.29	27.00	76.27
2011-10-12	402.19	26.96	77.16
2011-10-13	408.43	27.18	76.37
2011-10-14	422.00	27.27	78.11

2292 rows × 3 columns

- `rolling` operator: behaves similarly to `resample` and `groupby`. It can be called on a Series or DataFrame along with a window

```
close_px.AAPL.plot()
close_px.AAPL.rolling(250).mean().plot()
```



5.2.7: Moving window functions

- By default rolling functions require all of the values in the window to be non-NA

```
close_px.AAPL.rolling(250).mean()
```

```
2003-01-02          NaN
2003-01-03          NaN
2003-01-06          NaN
2003-01-07          NaN
2003-01-08          NaN
                  ...
2011-10-10    347.58772
2011-10-11    347.95668
2011-10-12    348.33412
2011-10-13    348.74688
2011-10-14    349.23096
Freq: B, Name: AAPL, Length: 2292, dtype: float64
```

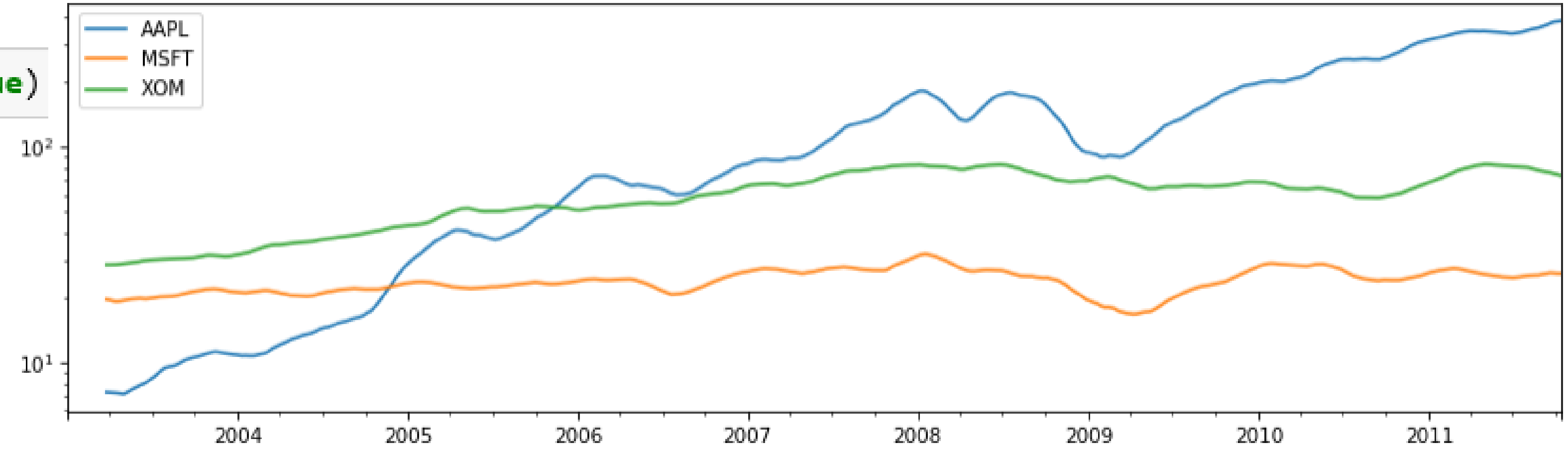
- Calling a moving window function on a DataFrame applies the transformation to each column

```
close_px.rolling(60).mean().plot(logy=True)
```

- `min_periods`:
  - Account for missing data
  - Fewer than window periods of data

```
appl_std250 = close_px.AAPL.rolling(250, min_periods=2).mean()
appl_std250
```

```
2003-01-02           NaN
2003-01-03      7.425000
2003-01-06      7.433333
2003-01-07      7.432500
2003-01-08      7.402000
                  ...
2011-10-10    347.587720
2011-10-11    347.956680
2011-10-12    348.334120
2011-10-13    348.746880
2011-10-14    349.230960
Freq: B, Name: AAPL, Length: 2292, dtype: float64
```

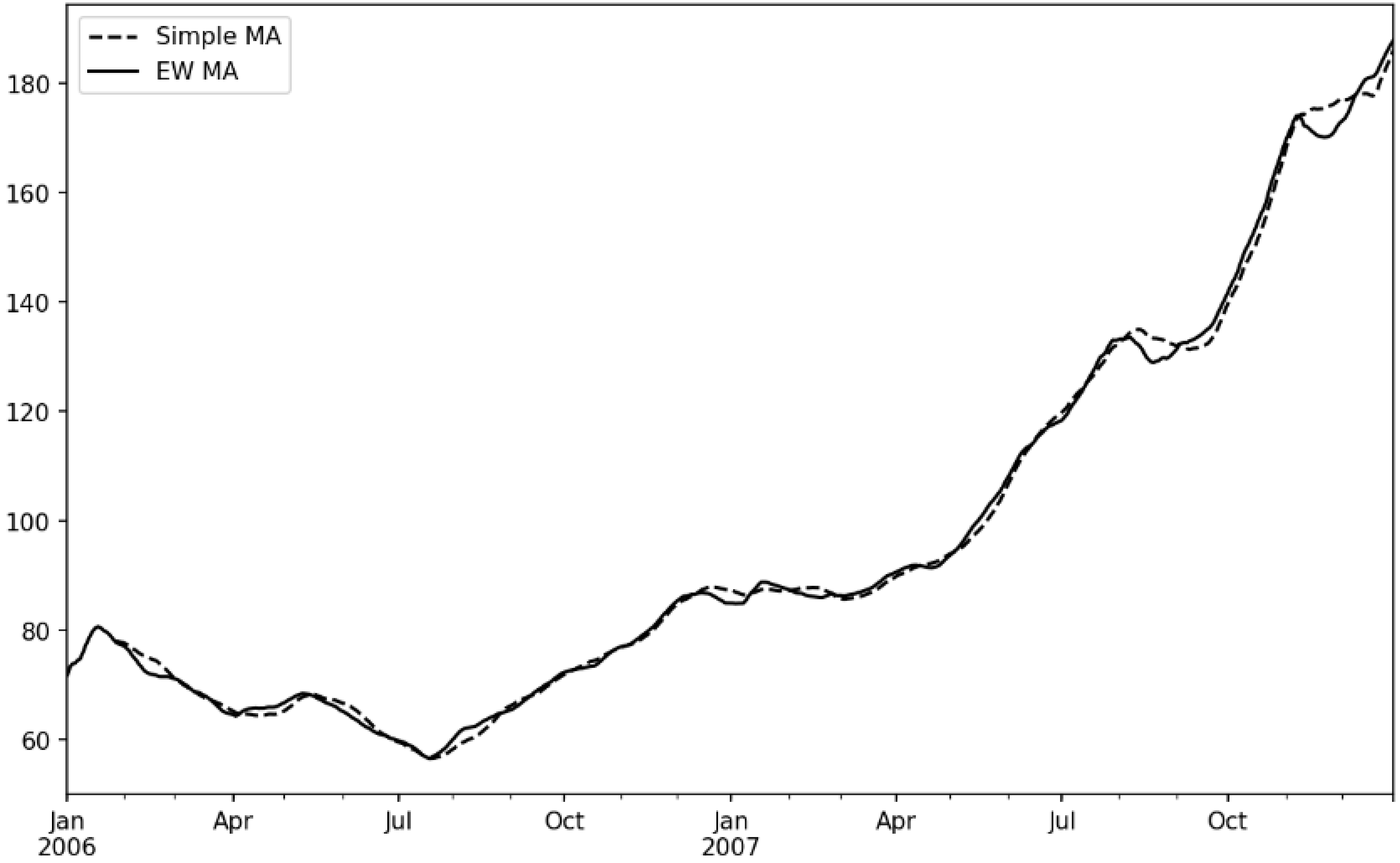


MAGIC CODE INSTITUTE
Leverage your tech skills

5.2.7: Moving window functions

- Exponentially Weighted Functions:
  - Adapt faster to recent changes
  - specify a constant *decay factor* to give more weight to more recent observations
  - `ewm` operator
  - `span` parameter

```
fig = plt.figure()
ma60  = close_px.AAPL.rolling(30, min_periods=20).mean()
ewma60 = close_px.AAPL.ewm(span=30).mean()
ma60.plot(style='k--', label='Simple MA')
ewma60.plot(style='k-', label='EW MA')
```

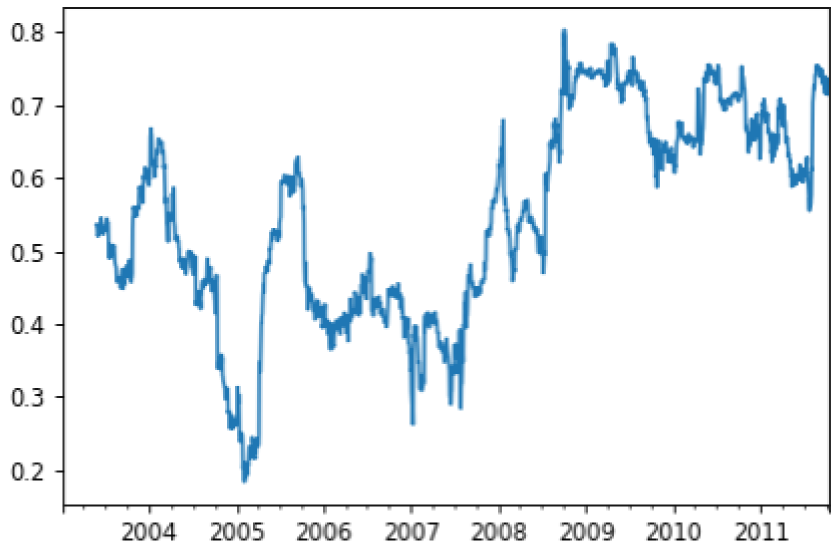


5.2.7: Moving window functions

- Some statistical operators, like correlation and covariance, need to operate on two time series.

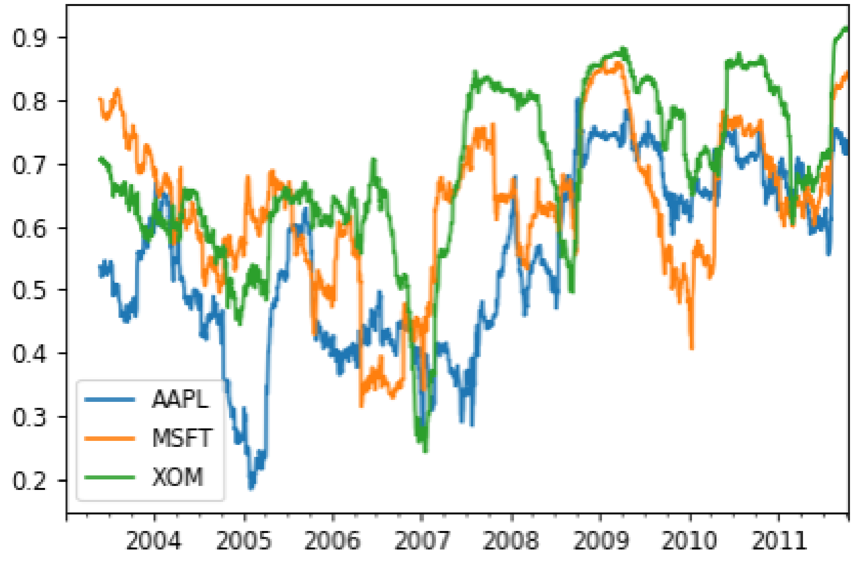
```
spx_px = close_px_all['SPX']
spx_rets = spx_px.pct_change()
returns = close_px.pct_change()
corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_rets)
fig = plt.figure()
corr.plot()
```

Figure 4




```
corr = returns.rolling(125, min_periods=100).corr(spx_rets)
corr.plot()
```

Figure 5



MAGIC CODE INSTITUTE

THANKS FOR LISTENING!!!