MCI

MAGIC CODE INSTITUTE

python™

Lecture 6:

Data Aggregation & Group Operations
Time Series Data

Section 6.1 | Data Aggregation & Group Operations

Section 6.1 | Data Aggregation & Group Operations

6.1.1: The GroupBy Mechanism

>> 6.1.1: The GroupBy Mechanism

>> DataFrame.groupby() method can be used in order to split DataFrame into groups and then aggregate data in each group

	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

	sepal_length	sepal_width	petal_length	petal_width
species				
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

>> 6.1.1: The GroupBy Mechanism

>> Prepare a DataFrame

```
import pandas as pd
import numpy as np

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	0.845636	-0.598147
1	a	two	-1.038174	-1.238751
2	b	one	0.082393	0.892178
3	b	two	1.357329	1.062574
4	a	one	-0.644777	0.508272

>> ***groupby*** always goes with an aggregate function such as ***sum()***, ***mean()*** etc.

>> ***groupby*** a single key value:

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

```
key1
a   -0.279105
b    0.719861
Name: data1, dtype: float64
```

>> ***groupby*** 2 key values

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

```
key1  key2
a     one     0.100429
      two    -1.038174
b     one     0.082393
      two     1.357329
Name: data1, dtype: float64
```

```
grouped.unstack()
```

key2 key1	one	two
a	0.100429	-1.038174
b	0.082393	1.357329

>> 6.1.1: The GroupBy Mechanism

MAGIC CODE INSTITUTE

>> For larger datasets, you might want to do one grouping and do aggregation one or more columns. This can easily be done with another paradigm:

```python
grouped = df.groupby(['key1', 'key2'])
grouped[['data1']].sum()
```

key1	key2	data1
a	one	-0.640659
	two	-0.240568
b	one	-0.055836
	two	0.235007

```python
grouped[['data2']].mean()
```

key1	key2	data2
a	one	0.277105
	two	1.590185
b	one	-0.347001
	two	-0.381311

>> If you don't want the return **DataFrame** with **MultiIndex** then specify *as_index=False* in the argument of groupby

```python
grouped = df.groupby(['key1', 'key2'], as_index=False)
grouped[['data1', 'data2']].sum()
```

	key1	key2	data1	data2
0	a	one	-0.640659	0.554210
1	a	two	-0.240568	1.590185
2	b	one	-0.055836	-0.347001
3	b	two	0.235007	-0.381311

>> 6.1.1: The GroupBy Mechanism

>> You can also pass a function as “column” for groupby, like this:

>> First let us create a people dataset:

```
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	-2.297238	-0.462857	-1.718973	-0.391475	0.901483
Steve	1.681930	0.114438	-0.331176	-0.117922	-0.155779
Wes	-0.316048	-1.632542	0.811137	0.751444	-0.546266
Jim	-0.632425	0.515077	-1.970805	0.548962	-0.420062
Travis	-0.285852	-0.596034	2.108096	-1.221094	-1.606421

>> Now we want to sum all value by the length of name value

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

	a	b	c	d	e
3	-3.245711	-1.580322	-2.878641	0.908932	-0.064845
5	1.681930	0.114438	-0.331176	-0.117922	-0.155779
6	-0.285852	-0.596034	2.108096	-1.221094	-1.606421

>> BUILT-IN 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

>> 6.1.1: The GroupBy Mechanism

>> You can define your own aggregate function

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

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

key1	data1	data2
a	1.883810	1.747023
b	1.274937	0.170396

>> Method like *describe()* also provides quick insight into the grouped dataset

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

key1	count	mean	std	min	25%	50%	75%	max
a	3.0	-0.279105	0.993716	-1.038174	-0.841476	-0.644777	0.100429	0.845636
b	2.0	0.719861	0.901516	0.082393	0.401127	0.719861	1.038595	1.357329

>> Pandas *groupby* also provides powerful functionality for Column-Wise groupby and Multiple Function aggregation

>> Let's revisit our ***tips.csv*** dataset

```
tips = pd.read_csv('tips.csv')
tips['tip_pct'] = tips['tip']*100/tips['total_bill']
tips
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	5.944673
1	10.34	1.66	No	Sun	Dinner	3	16.054159
2	21.01	3.50	No	Sun	Dinner	3	16.658734
3	23.68	3.31	No	Sun	Dinner	2	13.978041
4	24.59	3.61	No	Sun	Dinner	4	14.680765
...	...	...	...	...	...	...	...
239	29.03	5.92	No	Sat	Dinner	3	20.392697
240	27.18	2.00	Yes	Sat	Dinner	2	7.358352
241	22.67	2.00	Yes	Sat	Dinner	2	8.822232
242	17.82	1.75	No	Sat	Dinner	2	9.820426
243	18.78	3.00	No	Thur	Dinner	2	15.974441

244 rows × 7 columns

>> Now we want to calculate the average of *tips_pct* by *day* and *smoker*

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

```
day   smoker
Fri   No        15.165044
      Yes       17.478305
Sat   No        15.804766
      Yes       14.790607
Sun   No        16.011294
      Yes       18.725032
Thur  No        16.029808
      Yes       16.386327
Name: tip_pct, dtype: float64
```

>> 6.1.1: The GroupBy Mechanism

>> What if we want to do multiple aggregation on the same column?

```
grouped['tip_pct'].agg(['mean', 'std', min_max])
```

day	smoker	mean	std	min_max
Fri	No	15.165044	2.812295	6.734944
	Yes	17.478305	5.129267	15.992499
Sat	No	15.804766	3.976730	23.519300
	Yes	14.790607	6.137495	29.009476
Sun	No	16.011294	4.234723	19.322576
	Yes	18.725032	15.413424	64.468495
Thur	No	16.029808	3.877420	19.335021
	Yes	16.386327	3.938881	15.124046

>> Or multiple aggregation on multiple columns:

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

		tip_pct			total_bill		
		count	mean	max	count	mean	max
day	smoker						
Fri	No	4	15.165044	18.773467	4	18.420000	22.75
	Yes	15	17.478305	26.348039	15	16.813333	40.17
Sat	No	45	15.804766	29.198966	45	19.661778	48.33
	Yes	42	14.790607	32.573290	42	21.276667	50.81
Sun	No	57	16.011294	25.267250	57	20.506667	48.17
	Yes	19	18.725032	71.034483	19	24.120000	45.35
Thur	No	45	16.029808	26.631158	45	17.113111	41.19
	Yes	17	16.386327	24.125452	17	19.190588	43.11

>> 6.1.1: The GroupBy Mechanism

MAGIC CODE INSTITUTE

>> Suppose you want to apply different functions to different columns

>> pass a *dict* to *agg* that contain the mapping of functions and columns

```
grouped.agg({'tip': np.mean, 'total_bill': 'max'})
```

day	smoker	tip	total_bill
Fri	No	2.812500	22.75
	Yes	2.714000	40.17
Sat	No	3.102889	48.33
	Yes	2.875476	50.81
Sun	No	3.167895	48.17
	Yes	3.516842	45.35
Thur	No	2.673778	41.19
	Yes	3.030000	43.11

>> The *dict* can also contain different set of multiple functions for each column

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

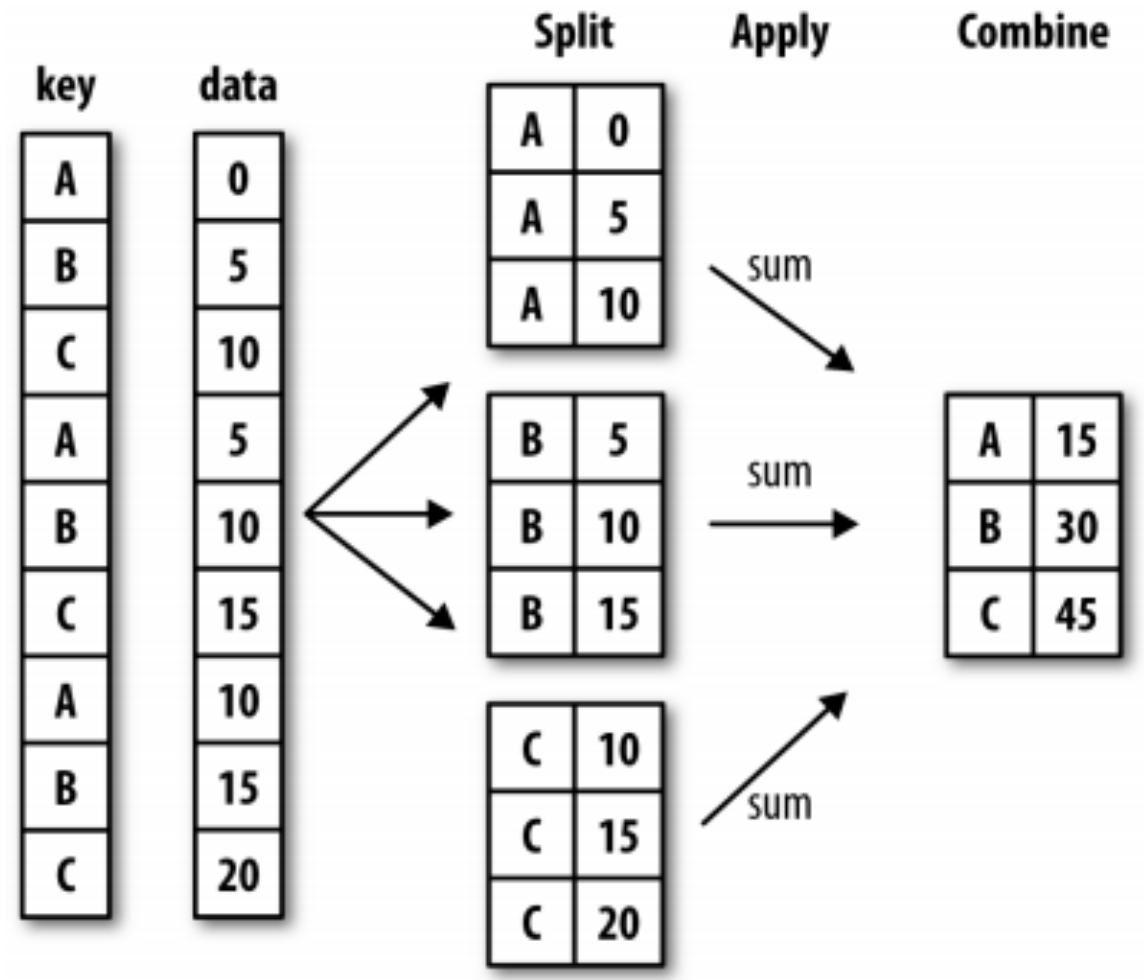
		tip_pct		size	
		mean	std	median	count
day	smoker				
Fri	No	15.165044	2.812295	2	4
	Yes	17.478305	5.129267	2	15
Sat	No	15.804766	3.976730	2	45
	Yes	14.790607	6.137495	2	42
Sun	No	16.011294	4.234723	3	57
	Yes	18.725032	15.413424	2	19
Thur	No	16.029808	3.877420	2	45
	Yes	16.386327	3.938881	2	17

Section 6.1 | Data Aggregation & Group Operations

6.1.2: Apply-Split and Apply-Combine

>> ***DataFrame.apply()*** splits the object being manipulated into segment (groups) and then invokes the passed function onto each segment, finally combine them back together.

>> ***apply*** in another sense is the most general-purpose ***groupby*** method.



>> 6.1.2: Apply-Split and Apply-Combine

>> Go back to our ***tips*** example

>> We want to select the top five tip_pct values by group

>> First, write a function that select the nth larget value in particular column

```
def top(df, n=5, column='tip_pct'):
    return df.sort_values(by=column, ascending=False).head(n)

top(tips)
```

	total_bill	tip	smoker	day	time	size	tip_pct
172	7.25	5.15	Yes	Sun	Dinner	2	71.034483
178	9.60	4.00	Yes	Sun	Dinner	2	41.666667
67	3.07	1.00	Yes	Sat	Dinner	1	32.573290
232	11.61	3.39	No	Sat	Dinner	2	29.198966
183	23.17	6.50	Yes	Sun	Dinner	4	28.053517

>> Group by *smoker* and call *apply* with this function. Let's see the result:

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

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	**232**	11.61	3.39	No	Sat	Dinner	2	29.198966
	149	7.51	2.00	No	Thur	Lunch	2	26.631158
	51	10.29	2.60	No	Sun	Dinner	2	25.267250
	185	20.69	5.00	No	Sun	Dinner	5	24.166264
	88	24.71	5.85	No	Thur	Lunch	2	23.674626
Yes	**172**	7.25	5.15	Yes	Sun	Dinner	2	71.034483
	178	9.60	4.00	Yes	Sun	Dinner	2	41.666667
	67	3.07	1.00	Yes	Sat	Dinner	1	32.573290
	183	23.17	6.50	Yes	Sun	Dinner	4	28.053517
	109	14.31	4.00	Yes	Sat	Dinner	2	27.952481

>> The *top* function was called on each row group of the DataFrame

>> The results are then glued together, labeling each segment with the group names thus the result has *hierarchical index* with the inner level contains index values from the original DataFrame

>> 6.1.2: Apply-Split and Apply-Combine

>> In order to pass arguments into the applied function, specify them after the function as keyword arguments *kwargs*:

```
tips.groupby('smoker').apply(top, n=3, column='total_bill')
```

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	212	48.33	9.00	No	Sat	Dinner	4	18.621974
	59	48.27	6.73	No	Sat	Dinner	4	13.942407
	156	48.17	5.00	No	Sun	Dinner	6	10.379905
Yes	170	50.81	10.00	Yes	Sat	Dinner	3	19.681165
	182	45.35	3.50	Yes	Sun	Dinner	3	7.717751
	102	44.30	2.50	Yes	Sat	Dinner	3	5.643341

>> Include *group_keys=False* in groupby in order to disable *hierarchical index*:

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

	total_bill	tip	smoker	day	time	size	tip_pct
232	11.61	3.39	No	Sat	Dinner	2	29.198966
149	7.51	2.00	No	Thur	Lunch	2	26.631158
51	10.29	2.60	No	Sun	Dinner	2	25.267250
185	20.69	5.00	No	Sun	Dinner	5	24.166264
88	24.71	5.85	No	Thur	Lunch	2	23.674626
172	7.25	5.15	Yes	Sun	Dinner	2	71.034483
178	9.60	4.00	Yes	Sun	Dinner	2	41.666667
67	3.07	1.00	Yes	Sat	Dinner	1	32.573290
183	23.17	6.50	Yes	Sun	Dinner	4	28.053517
109	14.31	4.00	Yes	Sat	Dinner	2	27.952481

>> Example of using groupby and apply to fill missing values with Group-Specific values

>> Let's create our dataset

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

```
Ohio          0.894882
New York      0.026810
Vermont       0.967012
Florida      -0.337318
Oregon        0.398596
Nevada       -1.380475
California   -2.020168
Idaho        -0.864595
dtype: float64
```

>> Add some missing value for our example

```
data[::2] = np.nan
data
```

```
Ohio               NaN
New York      0.026810
Vermont            NaN
Florida      -0.337318
Oregon             NaN
Nevada       -1.380475
California         NaN
Idaho        -0.864595
dtype: float64
```

>> Example of using groupby and apply to fill missing values with Group-Specific values

>> How we did?

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

```
Ohio         -0.638895
New York      0.026810
Vermont      -0.638895
Florida      -0.337318
Oregon       -0.638895
Nevada       -1.380475
California   -0.638895
Idaho        -0.864595
dtype: float64
```

>> Now we can do

```
data.groupby(coasts).mean()
```

```
East   -0.155254
West   -1.122535
dtype: float64
```

```
data.groupby(coasts).apply(lambda x: x.fillna(x.mean()))
```

```
Ohio         -0.155254
New York      0.026810
Vermont      -0.155254
Florida      -0.337318
Oregon       -1.122535
Nevada       -1.380475
California   -1.122535
Idaho        -0.864595
dtype: float64
```

Section 6.1 | Data Aggregation & Group Operations

6.1.3: Pivot Table

>> 6.1.3: Pivot Table

>> Pivot table with pandas in Python combines the *groupby* facility with reshape operations utilizing *hierarchical indexing*

>> The following 2 statements produce exactly the same result:

```
grouped = tips.groupby(['day', 'smoker']).mean()
pivoted = tips.pivot_table(index=['day', 'smoker']) # The default aggfunc=np.mean
display_side_by_side(grouped, pivoted)
```

day	smoker	total_bill	tip	size	tip_pct
Fri	No	18.420000	2.812500	2.250000	15.165044
	Yes	16.813333	2.714000	2.066667	17.478305
Sat	No	19.661778	3.102889	2.555556	15.804766
	Yes	21.276667	2.875476	2.476190	14.790607
Sun	No	20.506667	3.167895	2.929825	16.011294
	Yes	24.120000	3.516842	2.578947	18.725032
Thur	No	17.113111	2.673778	2.488889	16.029808
	Yes	19.190588	3.030000	2.352941	16.386327

day	smoker	size	tip	tip_pct	total_bill
Fri	No	2.250000	2.812500	15.165044	18.420000
	Yes	2.066667	2.714000	17.478305	16.813333
Sat	No	2.555556	3.102889	15.804766	19.661778
	Yes	2.476190	2.875476	14.790607	21.276667
Sun	No	2.929825	3.167895	16.011294	20.506667
	Yes	2.578947	3.516842	18.725032	24.120000
Thur	No	2.488889	2.673778	16.029808	17.113111
	Yes	2.352941	3.030000	16.386327	19.190588

>> 6.1.3: Pivot Table

>> For our ***tips*** example, suppose we want to aggregate only *tip_pct* and *size* and group by *time*.

>> *smoker* and *day* will be the inner level column and row:

```
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	13.962237	16.534736
	Sat	2.555556	2.476190	15.804766	14.790607
	Sun	2.929825	2.578947	16.011294	18.725032
	Thur	2.000000	NaN	15.974441	NaN
Lunch	Fri	3.000000	1.833333	18.773467	18.893659
	Thur	2.500000	2.352941	16.031067	16.386327

>> In order to have ***total*** and ***subtotal*** like in Excel, passing *margins*=*True* to the argument

```
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	13.962237	16.534736	15.891611
	Sat	2.555556	2.476190	2.517241	15.804766	14.790607	15.315172
	Sun	2.929825	2.578947	2.842105	16.011294	18.725032	16.689729
	Thur	2.000000	NaN	2.000000	15.974441	NaN	15.974441
Lunch	Fri	3.000000	1.833333	2.000000	18.773467	18.893659	18.876489
	Thur	2.500000	2.352941	2.459016	16.031067	16.386327	16.130074
All		2.668874	2.408602	2.569672	15.932846	16.319604	16.080258

>> 6.1.3: Pivot Table

>> In order to user different aggregate function

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

	day	Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	41.886710	711.214459	912.643775	15.974441	1681.719385
	Yes	148.812623	621.205474	355.775601	NaN	1125.793698
Lunch	No	18.773467	NaN	NaN	705.366927	724.140394
	Yes	113.361955	NaN	NaN	278.567563	391.929518
All		322.834755	1332.419933	1268.419376	999.908931	3923.582994

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

	day	Fri	Sat	Sun	Thur
time	smoker				
Dinner	No	1.784105	3.976730	4.234723	NaN
	Yes	5.267631	6.137495	15.413424	NaN
Lunch	No	NaN	NaN	NaN	3.922238
	Yes	5.026242	NaN	NaN	3.938881

Section 6.2 | Time-Series Data

Section 6.2 | Time-Series Data

6.2.1: Date and Time DataTypes and Tools

>> 6.2.1: Date and Time DataTypes and Tools

MAGIC CODE INSTITUTE

>> What is Time-Series?

"Anything that is observed or measured at many points in time."

Fixed frequency is time-series data with data points occur at regular intervals such as:

- every 15 seconds
- every 5 minutes
- once per month

>> Time-Series Data Type?

- ***timestamp***: specific instants in time usually in micro-second
- ***Fixed periods***: such as monthly Jan-2020 or yearly 2020
- ***interval of time***: indicated by a start and end timestamp. Periods can be thought of as special cases of intervals

>> 6.2.1: Date and Time DataTypes and Tools

MAGIC CODE INSTITUTE

>> *datetime* stores both the *date* and the *time* down to micro-second

```
from datetime import datetime

now = datetime.now()
now
```

```
datetime.datetime(2020, 8, 17, 16, 15, 46, 959999)
```

```
print(r'The date today is {}/{}/{}'.format(now.day, now.month, now.year))
```

```
The date today is 17/8/2020
```

>> *timedelta* represents the temporal difference between *datetime* objects

```
xmas_day = datetime(2020, 12, 24, 23, 59, 59)
time_til_xmas = xmas_day - datetime.today()
time_til_xmas
```

```
datetime.timedelta(days=129, seconds=27595, microseconds=138157)
```

```
message = 'It\'s approximately {} days {} hours till the Christmas! Hallelujah!'
print(message.format(time_til_xmas.days, round(time_til_xmas.seconds/3600)))
```

```
It's approximately 129 days 8 hours till the Christmas! Hallelujah!
```

>> Add or subtract *datetime* object by *timedelta*

```
from datetime import timedelta

to_add = 69
message = 'In {} days it will be {}'
added = datetime.today() + timedelta(days=to_add)
print(message.format(to_add, added))
```

```
In 69 days it will be 2020-10-25 16:28:23.052150
```

>> 6.2.1: Date and Time DataTypes and Tools

>> Format *datetime* for cleaner read

```
formated = added.strftime('%d-%b-%Y')
print(message.format(to_add, formated))
```

```
In 69 days it will be 25-Oct-2020
```

>> If you have a *datetime* string and want to convert it to object

```
datetime.strptime(formated, '%d-%b-%Y')
```

```
datetime.datetime(2020, 10, 25, 0, 0)
```

>> There are also the *dateutil* library that can simplify the *datetime* parsing

```
from dateutil.parser import parse

parse('2020-11-20')
```

```
datetime.datetime(2020, 11, 20, 0, 0)
```

>> As long as the *datetime* follow with standard format, no formater string is needed

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

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

```
# If date appears before month then
parse('16/08/2020', dayfirst=True)
```

```
datetime.datetime(2020, 8, 16, 0, 0)
```

Section 6.2 | Time-Series Data

6.2.2: Time-Series with Pandas

>> 6.2.2: Time-Series with Pandas

MAGIC CODE INSTITUTE

>> ***Pandas*** has built-in *datetime* method that you can utilize. The *pandas.to_datetime()* method will convert datetime string to object

```
date_strs = ['2020-08-16 12:00:00', '2020-08-17 00:00:00']
pd.to_datetime(date_strs)
```

```
DatetimeIndex(['2020-08-16 12:00:00', '2020-08-17 00:00:00'],
dtype='datetime64[ns]', freq=None)
```

>> It can also handle values that should be considered missing (None, empty string, etc.)

```
idx = pd.to_datetime([*date_strs, None])
idx
```

```
DatetimeIndex(['2020-08-16 12:00:00', '2020-08-17 00:00:00',
'NaT'], dtype='datetime64[ns]', freq=None)
```

```
idx[2]
```

```
NaT
```

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

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

>> 6.2.2: Time-Series with Pandas

Example*:* Create a pd.Series with datetime index

```
dates = [datetime(2020, 8, i) for i in range(1, 7)]
ts = pd.Series(np.random.randn(6), index=dates)
ts
```

```
2020-08-01   -0.629508
2020-08-02    0.272963
2020-08-03    0.933834
2020-08-04    1.458715
2020-08-05    1.577901
2020-08-06   -0.807677
dtype: float64
```

```
ts.index
```

```
DatetimeIndex(['2020-08-01', '2020-08-02', '2020-08-03', '2020-08-04',
               '2020-08-05', '2020-08-06'],
              dtype='datetime64[ns]', freq=None)
```

>> Indexing and Selection

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

```
Timestamp('2020-08-01 00:00:00')
```

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

```
-0.7108004784750609
```

```
ts['08/03/2020']
```

```
-0.7108004784750609
```

```
ts['20200803']
```

```
-0.7108004784750609
```

>> 6.2.2: Time-Series with Pandas

>> The Pandas *datetime* index is very useful especially with longer time-series

```
longer_ts = pd.Series(np.random.randn(1000),
                      index=pd.date_range('2001-09-11', periods=1000))
longer_ts
```

```
2001-09-11   -0.295558
2001-09-12   -0.064306
2001-09-13   -1.394690
2001-09-14   -0.257778
2001-09-15   -1.301347
                ...
2004-06-02    1.422945
2004-06-03   -0.544858
2004-06-04   -0.080557
2004-06-05   -0.229523
2004-06-06   -0.596775
Freq: D, Length: 1000, dtype: float64
```

>> Select data by year

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

```
2001-09-11   -0.295558
2001-09-12   -0.064306
2001-09-13   -1.394690
2001-09-14   -0.257778
2001-09-15   -1.301347
                ...
2001-12-27    0.253653
2001-12-28    1.482951
2001-12-29   -0.481669
2001-12-30   -0.235876
2001-12-31    0.702548
Freq: D, Length: 112, dtype: float64
```

>> Select data by year and month

```
longer_ts['2001-09']
```

```
2001-09-11   -0.295558
2001-09-12   -0.064306
2001-09-13   -1.394690
2001-09-14   -0.257778
2001-09-15   -1.301347
2001-09-16    1.297011
2001-09-17    0.225550
2001-09-18   -0.913104
2001-09-19    0.753292
2001-09-20    1.288387
2001-09-21   -0.265957
2001-09-22    0.534660
2001-09-23   -0.377165
2001-09-24   -0.133395
2001-09-25   -0.665442
2001-09-26    0.289807
2001-09-27    0.347251
2001-09-28   -0.835531
2001-09-29    1.037524
2001-09-30   -0.983144
Freq: D, dtype: float64
```

Section 6.2 | Time-Series Data

6.2.3: Generating DateRange, Frequency and Shifting

>> 6.2.3: Generating DateRange, Frequency and Shifting

MAGIC CODE INSTITUTE
Leverage your tech skills

>> Generating DateRange

```
pd.date_range('2001-09-11', periods=1000)
```

```
DatetimeIndex(['2001-09-11', '2001-09-12', '2001-09-13', '2001-09-14',
               '2001-09-15', '2001-09-16', '2001-09-17', '2001-09-18',
               '2001-09-19', '2001-09-20',
               ...
               '2004-05-28', '2004-05-29', '2004-05-30', '2004-05-31',
               '2004-06-01', '2004-06-02', '2004-06-03', '2004-06-04',
               '2004-06-05', '2004-06-06'],
              dtype='datetime64[ns]', length=1000, freq='D')
```

>> *date_range* can also be used to generate range with start, end and frequency

```
pd.date_range('2001-09-11', '2004-06-06', freq='BM')
```

```
DatetimeIndex(['2001-09-28', '2001-10-31', '2001-11-30', '2001-12-31',
               '2002-01-31', '2002-02-28', '2002-03-29', '2002-04-30',
               '2002-05-31', '2002-06-28', '2002-07-31', '2002-08-30',
               '2002-09-30', '2002-10-31', '2002-11-29', '2002-12-31',
               '2003-01-31', '2003-02-28', '2003-03-31', '2003-04-30',
               '2003-05-30', '2003-06-30', '2003-07-31', '2003-08-29',
               '2003-09-30', '2003-10-31', '2003-11-28', '2003-12-31',
               '2004-01-30', '2004-02-27', '2004-03-31', '2004-04-30',
               '2004-05-31'],
              dtype='datetime64[ns]', freq='BM')
```

>> date_range will preserve the time of timestamp

```
pd.date_range('2020-08-18 12:30:30', periods=5)
```

```
DatetimeIndex(['2020-08-18 12:30:30', '2020-08-19 12:30:30',
               '2020-08-20 12:30:30', '2020-08-21 12:30:30',
               '2020-08-22 12:30:30'],
              dtype='datetime64[ns]', freq='D')
```

>> Putting an integer as frequency multiplier

```
pd.date_range('2020-08-18', '2020-08-20', freq='4h')
```

```
DatetimeIndex(['2020-08-18 00:00:00', '2020-08-18 04:00:00',
               '2020-08-18 08:00:00', '2020-08-18 12:00:00',
               '2020-08-18 16:00:00', '2020-08-18 20:00:00',
               '2020-08-19 00:00:00', '2020-08-19 04:00:00',
               '2020-08-19 08:00:00', '2020-08-19 12:00:00',
               '2020-08-19 16:00:00', '2020-08-19 20:00:00',
               '2020-08-20 00:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

>> 6.2.3: Generating DateRange, Frequency and Shifting

>> Generating DateRange

>> *date_range* can also be used to generate range with start, end and frequency

>> date_range will preserve the time of timestamp

>> Putting an integer as frequency multiplier

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