Pandas datetime

In the last lesson, we explored the datetime is lesson will introduce datetime objects in Pandas.

Learning Objectives:

- Use pd.to_datetime & create a datetime index in pandas.
- Use date format codes for printing dates.
- Filter a time series data set.
- Find dates of extreme values using .idxmax() and .idxmin().

Panda's Datetime Functionality

So far, we have just been exploring datetime with the current date and time. Let's look at a data set that has a time series feature. We will be using the Daily Climate Time Series Dataset from Kaggle.

```
import pandas as pd
url="https://docs.google.com/spreadsheets/d/e/2PACX-1vQcpVvVioO23cndDwr1UmKhndrSq6ES6ZUKZ4fkBBqIAavd1_coVPO_yeOye-Ub-cAWlkX3psJvOU8o/pub?output=csv"
df = pd.read_csv(url)
df.info()
df.head(3)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1462 entries, 0 to 1461
Data columns (total 5 columns):
#   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
0   date          1462 non-null   object 
1   meantemp      1462 non-null   float64
2   humidity      1462 non-null   float64
3   wind_speed    1462 non-null   float64
4   meanpressure  1462 non-null   float64
dtypes: float64(4), object(1)
memory usage: 57.2+ KB
```

	date	meantemp	humidity	wind_speed	meanpressure
0	2013-01-01	10.000000	84.5	0.000000	1015.666667
1	2013-01-02	7.400000	92.0	2.980000	1017.800000
2	2013-01-03	7.166667	87.0	4.633333	1018.666667

- In the above dataset, we can see that we have a date column. According to our .info(), this column is still a string ("object" dtype)
- From our .head(), we can see that the date column seems to formatted as
  - 4-digit year, a dash, 2 digit month, a dash, and 2 digit day.
  - there is no time information, just the date.

pd.to_datetime

Pandas has a very helpful function called `to_datetime` that can intelligently convert a column into the "datetime" data type.

- - pd.to_datetime Documentation

Let's convert our date column to datetime and then check our .info() and .head() again. Note, that we are keeping the original "date" column as a string, and adding an additional column.

```
#Add a datetime column based on the string in the "date" column
df['datetime'] = pd.to_datetime(df['date'])
df.info()
df.head(3)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1462 entries, 0 to 1461
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
0   date          1462 non-null   object
1   meantemp      1462 non-null   float64
2   humidity      1462 non-null   float64
3   wind_speed    1462 non-null   float64
4   meanpressure  1462 non-null   float64
5   datetime      1462 non-null   datetime64[ns]
dtypes: datetime64[ns](1), float64(4), object(1)
memory usage: 68.7+ KB
```

	date	meantemp	humidity	wind_speed	meanpressure	datetime
0	2013-01-01	10.000000	84.5	0.000000	1015.666667	2013-01-01
1	2013-01-02	7.400000	92.0	2.980000	1017.800000	2013-01-02
2	2013-01-03	7.166667	87.0	4.633333	1018.666667	2013-01-03

We can see that our new "datetime" column's dtype is listed as "datetime64[ns]" and the original date is still a string.

- Note: [ns] is indicating nano-second precision.
- Notice, however, that the values stored in the date and datetime columns both *look* the same in the output!
    - This is misleading, because when viewed as a dataframe or series, the output will always *show* the string version of the datetime even if it is actually a datetime object.
- Let's slice out 1 value from both columns with .loc and see how the values look when viewed alone.

```
## preview first row from the date column (string)
df.loc[0,'date']
```

```
'2013-01-01'
```

```
## preview first row from the datetime column (datetime64[ns])
df.loc[0,'datetime']
```

```
Timestamp('2013-01-01 00:00:00')
```

- When viewed single objects, NOW we can see the difference. The values stored in our new 'datetime' column are a special new type of object called a Timestamp.
- The Timestamp class is the simplest datetime variable type in Pandas and shares much of the same functionality as the basic Python datetime class.
- We can use .strftime with Pandas Timestamps just as we demonstrated with basic Python datetime class in the previous lesson. The formatting codes will also be the same.

For example,

- If you wanted "January 01, 2013", you would use: `"%B %d, %Y".`

```
## demonstrate format code
fmt = "%B %d, %Y"
df.loc[0,'datetime'].strftime(fmt)
```

```
'January 01, 2013'
```

Using the `.dt.` accessor for a datetime column

- Just like we can slice out a string column and use `.str.` to access string methods, Pandas also has a `.dt.` for datetime columns.

.dt.strftime

Here we will apply the desired format and create an entire new column:

```
df['datetime_fmt'] = df['datetime'].dt.strftime(fmt)
df['datetime_fmt']
```

```
0        January 01, 2013
1        January 02, 2013
2        January 03, 2013
3        January 04, 2013
4        January 05, 2013
              ...
1457    December 28, 2016
1458    December 29, 2016
1459    December 30, 2016
1460    December 31, 2016
1461     January 01, 2017
Name: datetime_1, Length: 1462, dtype: object
```

- Important Note: .dt.strftime returns a STRING not an actual datetime column. (notice in the output below that the "datetime_fmt" column is an "object" dtype)

```
df.info()
```

```
<class 'pandas.core.frame.dataframe'="">
RangeIndex: 1462 entries, 0 to 1461
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   date          1462 non-null   object
 1   meantemp      1462 non-null   float64
 2   humidity      1462 non-null   float64
 3   wind_speed    1462 non-null   float64
 4   meanpressure  1462 non-null   float64
 5   datetime      1462 non-null   datetime64[ns]
 6   datetime_fmt  1462 non-null   object
dtypes: datetime64[ns](1), float64(4), object(2)
memory usage: 80.1+ KB</class>
```

Using .dt.strftime is fine for when we simply want to *display* dates, but when we actually want to continue to use the feature as a datetime column, we must accept the default formatting.

The good news is that pandas is very flexible in how WE format our dates when slicing a datetime column. This is demonstrated below.

.dt.year/month/day/hour/etc

With the datetime object, we have a great deal of flexibility in accessing the specific information we want. For example, we could obtain just the year.

```
df['datetime'].dt.year.head()
```

```
0    2013
1    2013
2    2013
3    2013
4    2013
Name: datetime, dtype: int64
```

To clarify, if we were to try this on our "datetime_fmt" column (which is just a string), it will not work! Try the following code to demonstrate the value of using the datetime object!

```
# Attempting to obtain year from a .dt.strftime object will NOT work!
df['datetime_fmt'].dt.year.head()
```

Error!

Some other examples:

We could obtain just the month:

```
df['datetime'].dt.month.head()
```

```
0    1
1    1
2    1
3    1
4    1
Name: datetime, dtype: int64
```

In some cases, we might be more interested in the quarter:

```
## fiscal year quarter
df['datetime'].dt.quarter.head()
```

```
0    1
1    1
2    1
3    1
4    1
Name: datetime, dtype: int64
```

We could also just focus on the day of the week:

```
## day of the week - numeric
df['datetime'].dt.day_of_week.head()
```

```
0       1
1       2
2       3
3       4
4       5
Name: datetime, dtype: int64
```

Or, if we prefer to have the name instead of the number:

```
## day of the week - String name
df['datetime'].dt.day_name().head()
```

```
0         Tuesday
1       Wednesday
2        Thursday
3          Friday
4        Saturday
Name: datetime, dtype: object
```

Setting a Datetime Index

Before we can start slicing our data using the datetime object, we will need to make the datetime column the index for the DataFrame or Series.

```
df = df.set_index('datetime')
df
```

datetime	date	meantemp	humidity	wind_speed	meanpressure	datetime_fmt
2013-01-01	2013-01-01	10.000000	84.500000	0.000000	1015.666667	January 01, 2013
2013-01-02	2013-01-02	7.400000	92.000000	2.980000	1017.800000	January 02, 2013
2013-01-03	2013-01-03	7.166667	87.000000	4.633333	1018.666667	January 03, 2013
2013-01-04	2013-01-04	8.666667	71.333333	1.233333	1017.166667	January 04, 2013
2013-01-05	2013-01-05	6.000000	86.833333	3.700000	1016.500000	January 05, 2013
...	...	...	...	...	...	...
2016-12-28	2016-12-28	17.217391	68.043478	3.547826	1015.565217	December 28, 2016
2016-12-29	2016-12-29	15.238095	87.857143	6.000000	1016.904762	December 29, 2016
2016-12-30	2016-12-30	14.095238	89.666667	6.266667	1017.904762	December 30, 2016
2016-12-31	2016-12-31	15.052632	87.000000	7.325000	1016.100000	December 31, 2016
2017-01-01	2017-01-01	10.000000	100.000000	0.000000	1016.000000	January 01, 2017

1462 rows × 6 columns

If we check the DataFrame's .index, we can confirm that it is indeed still datetime dtype.

```
df.index
```

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06', '2013-01-07', '2013-01-08',
               '2013-01-09', '2013-01-10',
               ...
               '2016-12-23', '2016-12-24', '2016-12-25', '2016-12-26',
               '2016-12-27', '2016-12-28', '2016-12-29', '2016-12-30',
               '2016-12-31', '2017-01-01'],
              dtype='datetime64[ns]', name='datetime', length=1462, freq=None)
```

Slicing a Time Series with pandas

- Pandas is incredibly flexible/generous in processing dates when we use `.loc` .
- For example, if we wanted every row from the year 2013, we can just use .loc and a string version of the year

```
df.loc['2013']
```

datetime	date	meantemp	humidity	wind_speed	meanpressure
2013-01-01	2013-01-01	10.000000	84.500000	0.000000	1015.666667
2013-01-02	2013-01-02	7.400000	92.000000	2.980000	1017.800000
2013-01-03	2013-01-03	7.166667	87.000000	4.633333	1018.666667
2013-01-04	2013-01-04	8.666667	71.333333	1.233333	1017.166667
2013-01-05	2013-01-05	6.000000	86.833333	3.700000	1016.500000
...	...	...	...	...	...
2013-12-27	2013-12-27	11.875000	79.875000	1.162500	1018.625000
2013-12-28	2013-12-28	10.875000	70.000000	5.325000	1019.250000
2013-12-29	2013-12-29	10.571429	69.428571	5.325000	1018.500000
2013-12-30	2013-12-30	12.375000	79.500000	6.475000	1018.125000
2013-12-31	2013-12-31	14.500000	89.375000	4.862500	1020.500000

365 rows × 5 columns

- We can also slice a range of dates, for example, all of 2013 and 2014:

```
df.loc['2013':'2014']
```

datetime	date	meantemp	humidity	wind_speed	meanpressure
2013-01-01	2013-01-01	10.000000	84.500000	0.000000	1015.666667
2013-01-02	2013-01-02	7.400000	92.000000	2.980000	1017.800000
2013-01-03	2013-01-03	7.166667	87.000000	4.633333	1018.666667
2013-01-04	2013-01-04	8.666667	71.333333	1.233333	1017.166667
2013-01-05	2013-01-05	6.000000	86.833333	3.700000	1016.500000
...	...	...	...	...	...
2014-12-27	2014-12-27	10.375000	69.000000	2.775000	1018.625000
2014-12-28	2014-12-28	9.000000	86.000000	0.700000	1019.750000
2014-12-29	2014-12-29	11.125000	72.625000	1.387500	1017.250000
2014-12-30	2014-12-30	11.625000	70.625000	2.550000	1014.625000
2014-12-31	2014-12-31	12.375000	67.125000	2.787500	1016.875000

730 rows × 5 columns

- It is also flexible with the frequency/specificity of the dates that we use.
- For example, if we wanted all of 2013 and everything up through June of 2014, we could use:

```
df.loc['2013':'06-2014']
```

datetime	date	meantemp	humidity	wind_speed	meanpressure
2013-01-01	2013-01-01	10.000000	84.500000	0.000000	1015.666667
2013-01-02	2013-01-02	7.400000	92.000000	2.980000	1017.800000
2013-01-03	2013-01-03	7.166667	87.000000	4.633333	1018.666667
2013-01-04	2013-01-04	8.666667	71.333333	1.233333	1017.166667
2013-01-05	2013-01-05	6.000000	86.833333	3.700000	1016.500000
...	...	...	...	...	...
2014-06-26	2014-06-26	32.500000	60.375000	3.237500	1001.000000
2014-06-27	2014-06-27	34.750000	48.500000	9.487500	1001.625000
2014-06-28	2014-06-28	34.875000	49.000000	5.787500	999.375000
2014-06-29	2014-06-29	33.000000	57.750000	9.712500	997.750000
2014-06-30	2014-06-30	32.000000	67.750000	3.250000	997.500000

546 rows × 5 columns

- Pandas even let's us use completely different date formats! (though this would get confusing in your own code and should be avoided).
- To get everything from 03/2013 to 06/2014 we could use:

```
# Just to demonstrate the flexibility, we can use different formats in one line of
code
df.loc['March, 2013':'06-2014']
```

datetime	date	meantemp	humidity	wind_speed	meanpressure
2013-03-01	2013-03-01	17.333333	49.333333	24.066667	1016.333333
2013-03-02	2013-03-02	19.000000	54.000000	15.725000	1016.250000
2013-03-03	2013-03-03	19.333333	62.833333	8.633333	1016.166667
2013-03-04	2013-03-04	17.600000	71.000000	5.560000	1015.800000
2013-03-05	2013-03-05	20.875000	61.875000	4.162500	1016.375000
...	...	...	...	...	...
2014-06-26	2014-06-26	32.500000	60.375000	3.237500	1001.000000
2014-06-27	2014-06-27	34.750000	48.500000	9.487500	1001.625000
2014-06-28	2014-06-28	34.875000	49.000000	5.787500	999.375000
2014-06-29	2014-06-29	33.000000	57.750000	9.712500	997.750000
2014-06-30	2014-06-30	32.000000	67.750000	3.250000	997.500000

487 rows × 5 columns

Finding dates of extreme values - using .idxmax() and .idxmin()

- Now that we have a datetime index, we can start answering some questions about historical extreme values.
  - For example, print the following sentence for the most humid day in in this dataset:
  - "The most humid day was {date as dd/mm/yyyy}, which was a {day of the week}. The humidity was { max humidity}, and the average temperature was {}"

Option 1: Find max and filter (not recommended)

- One approach we could take is to find the highest humidity value , we could use `df['humidity'].max()` , but that would only give us the humidity value.
    - We will also have to do some filtering with pandas to find the date with the max:

```
## using max and slicing to find the index
max_humid = df['humidity'].max()
most_humid_date = df.loc[ df['humidity']==max_humid]
max_date = most_humid_date.index
most_humid_date
```

	date	meantemp	humidity	wind_speed	meanpressure
datetime					
2017-01-01	2017-01-01	10.0	100.0	0.0	1016.0

```
## now print the requested statement
print(f"The most humid day was {max_date.strftime('%m/%d/%Y')}, which was a
{max_date.day_name()}.")
print(f" The humidity was {max_humid}, and the average temperature was
{most_humid_date['meantemp']}")
```

```
The most humid day was Index(['01/01/2017'], dtype='object', name='datetime'),
which was a Index(['Sunday'], dtype='object', name='datetime'). The humidity was
100.0, and the average temperature was datetime
2017-01-01    10.0
Name: meantemp, dtype: float64
```

- As you can see, we receive Pandas series for most of the values we sliced, even though we only had 1 row.

Option 2: Using .idxmax() (recommended)

- An alternative (and better) option:
    - We can use the `.idxmax()` method to get the INDEX of the maximum value.

```
max_date = df['humidity'].idxmax()
max_date
```

```
Timestamp('2017-01-01 00:00:00')
```

```
df.loc[max_date]
```

```
date                  2017-01-01
meantemp                    10.0
humidity                   100.0
wind_speed                   0.0
meanpressure              1016.0
Name: 2017-01-01 00:00:00, dtype: object
```

We need to construct a print statement that will use several pieces of information that we want to pull from our Series for the max_date.

- To simplify the f-string we will create, we can save all of the values from our 1 row of 5 values (date, meantemp, humidity, wind_speed, meanpressure) in separate variables. The long/tedious way we could do this is to use a separate line for each variable we want to create and use a `df.loc` to slice out the right feature.

```
## TEDIOUS WAY/BAD WAY
## Saving each value from the series as separate variables
str_date = df.loc[max_date,'date']
temp = df.loc[max_date,'meantemp']
humidity = df.loc[max_date,'humidity']
wind = df.loc[max_date,'wind_speed']
pressure = df.loc[max_date,'meanpressure']
# Proving that it worked!
str_date, temp, humidity, wind, pressure
```

```
('2017-01-01', 10.0, 100.0, 0.0, 1016.0)
```

- Instead of having to do this in many lines, we can do it all in 1 line by leveraging Python's tuple unpacking.
    - See this Geeks for Geeks article for more information on unpacking with python.
- In short, if we have a variable that contains several values, we can slice out each value into a new variable by setting a variable name for each variable we want to create on the left side of the `=` and then the container (list/series/etc) that we want to unpack on the right side of the `=`.

```
df.loc[max_date]
```

```
date                  2017-01-01
meantemp                    10.0
humidity                   100.0
wind_speed                   0.0
meanpressure              1016.0
Name: 2017-01-01 00:00:00, dtype: object
```

```
## Saving each value from the series as separate variables
str_date,temp, humidity, wind, pressure = df.loc[max_date]
# Proving that it worked!
str_date, temp, humidity, wind, pressure
```

```
('2017-01-01', 10.0, 100.0, 0.0, 1016.0)
```

```
## now print the requested statement
print(f"The most humid day was {max_date.strftime('%m/%d/%Y')}, \
which was a {max_date.day_name()}.")
print(f"The humidity was {humidity:.2f},\
and the average temperature was {temp:.2f}.")
```

Previous

Next