DateFunctionality_ed

May 30, 2022

In today's lecture, where we'll be looking at the time series and date functionally in pandas. Manipulating dates and time is quite flexible in Pandas and thus allows us to conduct more analysis such as time series analysis, which we will talk about soon. Actually, pandas was originally created by Wed McKinney to handle date and time data when he worked as a consultant for hedge funds.

[1]:
```
# Let's bring in pandas and numpy as usual
import pandas as pd
import numpy as np
```

0.0.1 Timestamp

[2]:
```
# Pandas has four main time related classes. Timestamp, DatetimeIndex, Period,␣
 ↪and PeriodIndex. First, let's
# look at Timestamp. It represents a single timestamp and associates values␣
 ↪with points in time.

# For example, let's create a timestamp using a string 9/1/2019 10:05AM, and␣
 ↪here we have our timestamp.
# Timestamp is interchangeable with Python's datetime in most cases.
pd.Timestamp('9/1/2019 10:05AM')
```

[2]:
```
Timestamp('2019-09-01 10:05:00')
```

[3]:
```
# We can also create a timestamp by passing multiple parameters such as year,␣
 ↪month, date, hour,
# minute, separately
pd.Timestamp(2019, 12, 20, 0, 0)
```

[3]:
```
Timestamp('2019-12-20 00:00:00')
```

[4]:
```
# Timestamp also has some useful attributes, such as isoweekday(), which shows␣
 ↪the weekday of the timestamp
# note that 1 represents Monday and 7 represents Sunday
pd.Timestamp(2019, 12, 20, 0, 0).isoweekday()
```

[4]:
```
5
```

[5]:
```
# You can find extract the specific year, month, day, hour, minute, second from␣
 ↪a timestamp
```

```
pd.Timestamp(2019, 12, 20, 5, 2,23).second
```

[5]: 23

0.0.2 Period

[6]:
```
# Suppose we weren't interested in a specific point in time and instead wanted␣
 ↪a span of time. This is where
# the Period class comes into play. Period represents a single time span, such␣
 ↪as a specific day or month.

# Here we are creating a period that is January 2016,
pd.Period('1/2016')
```

[6]: Period('2016-01', 'M')

[7]:
```
# You'll notice when we print that out that the granularity of the period is M␣
 ↪for month, since that was the
# finest grained piece we provided. Here's an example of a period that is March␣
 ↪5th, 2016.
pd.Period('3/5/2016')
```

[7]: Period('2016-03-05', 'D')

[8]:
```
# Period objects represent the full timespan that you specify. Arithmetic on␣
 ↪period is very easy and
# intuitive, for instance, if we want to find out 5 months after January 2016,␣
 ↪we simply plus 5
pd.Period('1/2016') + 5
```

[8]: Period('2016-06', 'M')

[9]:
```
# From the result, you can see we get June 2016. If we want to find out two␣
 ↪days before March 5th 2016, we
# simply subtract 2
pd.Period('3/5/2016') - 2
```

[9]: Period('2016-03-03', 'D')

[10]:
```
# The key here is that the period object encapsulates the granularity for␣
 ↪arithmetic
```

0.0.3 DatetimeIndex and PeriodIndex

[11]:
```
# The index of a timestamp is DatetimeIndex. Let's look at a quick example.␣
 ↪First, let's create our example
# series t1, we'll use the Timestamp of September 1st, 2nd and 3rd of 2016.␣
 ↪When we look at the series, each
# Timestamp is the index and has a value associated with it, in this case, a, b␣
 ↪and c.
```

```
t1 = pd.Series(list('abc'), [pd.Timestamp('2016-09-01'), pd.
 ↪Timestamp('2016-09-02'),
                              pd.Timestamp('2016-09-03')])
t1
```

[11]:
```
2016-09-01    a
2016-09-02    b
2016-09-03    c
dtype: object
```

[12]:
```
# Looking at the type of our series index, we see that it's DatetimeIndex.
type(t1.index)
```

[12]:
```
pandas.core.indexes.datetimes.DatetimeIndex
```

[13]:
```
# Similarly, we can create a period-based index as well.
t2 = pd.Series(list('def'), [pd.Period('2016-09'), pd.Period('2016-10'),
                              pd.Period('2016-11')])
t2
```

[13]:
```
2016-09    d
2016-10    e
2016-11    f
Freq: M, dtype: object
```

[14]:
```
# Looking at the type of the ts2.index, we can see that it's PeriodIndex.
type(t2.index)
```

[14]:
```
pandas.core.indexes.period.PeriodIndex
```

0.0.4 Converting to Datetime

[15]:
```
# Now, let's look into how to convert to Datetime. Suppose we have a list of␣
 ↪dates as strings and we want to
# create a new dataframe

# I'm going to try a bunch of different date formats
d1 = ['2 June 2013', 'Aug 29, 2014', '2015-06-26', '7/12/16']

# And just some random data
ts3 = pd.DataFrame(np.random.randint(10, 100, (4,2)), index=d1,
                   columns=list('ab'))
ts3
```

[15]:

	a	b
2 June 2013	38	40
Aug 29, 2014	94	99
2015-06-26	75	32
7/12/16	68	62

[16]:
```
# Using pandas to_datetime, pandas will try to convert these to Datetime and␣
 ↪put them in a standard format.

ts3.index = pd.to_datetime(ts3.index)
ts3
```

[16]:
```
              a   b
2013-06-02  38  40
2014-08-29  94  99
2015-06-26  75  32
2016-07-12  68  62
```

[17]:
```
# to_datetime also() has options to change the date parse order. For example,␣
 ↪we
# can pass in the argument dayfirst = True to parse the date in European date.

pd.to_datetime('4.7.12', dayfirst=True)
```

[17]:
```
Timestamp('2012-07-04 00:00:00')
```

0.0.5 Timedelta

[18]:
```
# Timedeltas are differences in times. This is not the same as a a period, but␣
 ↪conceptually similar. For
# instance, if we want to take the difference between September 3rd and ␣
 ↪September 1st, we get a Timedelta of
# two days.
pd.Timestamp('9/3/2016')-pd.Timestamp('9/1/2016')
```

[18]:
```
Timedelta('2 days 00:00:00')
```

[19]:
```
# We can also do something like find what the date and time is for 12 days and␣
 ↪three hours past September 2nd,
# at 8:10 AM.
pd.Timestamp('9/2/2016 8:10AM') + pd.Timedelta('12D 3H')
```

[19]:
```
Timestamp('2016-09-14 11:10:00')
```

0.0.6 Offset

[20]:
```
# Offset is similar to timedelta, but it follows specific calendar duration␣
 ↪rules. Offset allows flexibility
# in terms of types of time intervals. Besides hour, day, week, month, etc it␣
 ↪also has business day, end of
# month, semi month begin etc

# Let's create a timestamp, and see what day is that
pd.Timestamp('9/4/2016').weekday()
```

[20]:
```
6
```

[21]:
```
# Now we can now add the timestamp with a week ahead
pd.Timestamp('9/4/2016') + pd.offsets.Week()
```

[21]:
```
Timestamp('2016-09-11 00:00:00')
```

[22]:
```
# Now let's try to do the month end, then we would have the last day of␣
 ↪Septemer
pd.Timestamp('9/4/2016') + pd.offsets.MonthEnd()
```

[22]:
```
Timestamp('2016-09-30 00:00:00')
```

0.0.7 Working with Dates in a Dataframe

[23]:
```
# Next, let's look at a few tricks for working with dates in a DataFrame.␣
 ↪Suppose we want to look at nine
# measurements, taken bi-weekly, every Sunday, starting in October 2016. Using␣
 ↪date_range, we can create this
# DatetimeIndex. In data_range, we have to either specify the start or end date.
 ↪ If it is not explicitly
# specified, by default, the date is considered the start date. Then we have to␣
 ↪specify number of periods, and
# a frequency. Here, we set it to "2W-SUN", which means biweekly on Sunday

dates = pd.date_range('10-01-2016', periods=9, freq='2W-SUN')
dates
```

[23]:
```
DatetimeIndex(['2016-10-02', '2016-10-16', '2016-10-30', '2016-11-13',
               '2016-11-27', '2016-12-11', '2016-12-25', '2017-01-08',
               '2017-01-22'],
              dtype='datetime64[ns]', freq='2W-SUN')
```

[24]:
```
# There are many other frequencies that you can specify. For example, you can␣
 ↪do business day
pd.date_range('10-01-2016', periods=9, freq='B')
```

[24]:
```
DatetimeIndex(['2016-10-03', '2016-10-04', '2016-10-05', '2016-10-06',
               '2016-10-07', '2016-10-10', '2016-10-11', '2016-10-12',
               '2016-10-13'],
              dtype='datetime64[ns]', freq='B')
```

[25]:
```
# Or you can do quarterly, with the quarter start in June
pd.date_range('04-01-2016', periods=12, freq='QS-JUN')
```

[25]:
```
DatetimeIndex(['2016-06-01', '2016-09-01', '2016-12-01', '2017-03-01',
               '2017-06-01', '2017-09-01', '2017-12-01', '2018-03-01',
               '2018-06-01', '2018-09-01', '2018-12-01', '2019-03-01'],
              dtype='datetime64[ns]', freq='QS-JUN')
```

[26]:
```
# Now, let's go back to our weekly on Sunday example and create a DataFrame␣
 ↪using these dates, and some random
# data, and see what we can do with it.

dates = pd.date_range('10-01-2016', periods=9, freq='2W-SUN')
df = pd.DataFrame({'Count 1': 100 + np.random.randint(-5, 10, 9).cumsum(),
                   'Count 2': 120 + np.random.randint(-5, 10, 9)}, index=dates)
df
```

[26]:

	Count 1	Count 2
2016-10-02	109	119
2016-10-16	109	127
2016-10-30	117	118
2016-11-13	115	129
2016-11-27	113	122
2016-12-11	122	129
2016-12-25	127	129
2017-01-08	135	117
2017-01-22	138	128

[27]:
```
# First, we can check what day of the week a specific date is. For example,␣
 ↪here we can see that all the dates
# in our index are on a Sunday. Which matches the frequency that we set
df.index.weekday_name
```

[27]:
```
Index(['Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday',
       'Sunday', 'Sunday'],
      dtype='object')
```

[28]:
```
# We can also use diff() to find the difference between each date's value.
df.diff()
```

[28]:

	Count 1	Count 2
2016-10-02	NaN	NaN
2016-10-16	0.0	8.0
2016-10-30	8.0	-9.0
2016-11-13	-2.0	11.0
2016-11-27	-2.0	-7.0
2016-12-11	9.0	7.0
2016-12-25	5.0	0.0
2017-01-08	8.0	-12.0
2017-01-22	3.0	11.0

[29]:
```
# Suppose we want to know what the mean count is for each month in our␣
 ↪DataFrame. We can do this using
# resample. Converting from a higher frequency from a lower frequency is called␣
 ↪downsampling (we'll talk about
# this in a moment)
df.resample('M').mean()
```

[29]:
```
               Count 1     Count 2
2016-10-31  111.666667  121.333333
2016-11-30  114.000000  125.500000
2016-12-31  124.500000  129.000000
2017-01-31  136.500000  122.500000
```

[30]:
```
# Now let's talk about datetime indexing and slicing, which is a wonderful␣
 ↪feature of the pandas DataFrame.
# For instance, we can use partial string indexing to find values from a␣
 ↪particular year,
df['2017']
```

[30]:
```
            Count 1  Count 2
2017-01-08      135      117
2017-01-22      138      128
```

[31]:
```
# Or we can do it from a particular month
df['2016-12']
```

[31]:
```
            Count 1  Count 2
2016-12-11      122      129
2016-12-25      127      129
```

[32]:
```
# Or we can even slice on a range of dates For example, here we only want the␣
 ↪values from December 2016
# onwards.
df['2016-12':]
```

[32]:
```
            Count 1  Count 2
2016-12-11      122      129
2016-12-25      127      129
2017-01-08      135      117
2017-01-22      138      128
```

[33]:
```
df['2016']
```

[33]:
```
            Count 1  Count 2
2016-10-02      109      119
2016-10-16      109      127
2016-10-30      117      118
2016-11-13      115      129
2016-11-27      113      122
2016-12-11      122      129
2016-12-25      127      129
```