Reading date and time data in Pandas

WORKING WITH DATES AND TIMES IN PYTHON

Max Shron
Data Scientist and Author

datacamp

A simple Pandas example

```
# Load Pandas
import pandas as pd
# Import W20529's rides in Q4 2017
rides = pd.read_csv('capital-onebike.csv')
```

A simple Pandas example

```
# See our data
print(rides.head(3))
```

```
            Start date             End date                    Start station  \
0 2017-10-01 15:23:25 2017-10-01 15:26:26              Glebe Rd & 11th St N
1 2017-10-01 15:42:57 2017-10-01 17:49:59  George Mason Dr & Wilson Blvd
2 2017-10-02 06:37:10 2017-10-02 06:42:53  George Mason Dr & Wilson Blvd

                           End station Bike number Member type
0         George Mason Dr & Wilson Blvd      W20529      Member
1         George Mason Dr & Wilson Blvd      W20529      Casual
2  Ballston Metro / N Stuart & 9th St N      W20529      Member
```

A simple Pandas example

```
rides['Start date']
```

```
0     2017-10-01 15:23:25
1     2017-10-01 15:42:57
...
Name: Start date, Length: 290, dtype: object
```

4. A simple Pandas example
We can also select a particular column by using the brackets, as here where we call rides['Start date']. And we can get a particular row with .iloc[], in this case row number 2. Because we didn't tell Pandas to treat the start date and end date columns as datetimes, they are simply strings or objects. We want them to be datetimes so we can work with them effectively, using the tools from the first three chapters of this course.

```
rides.iloc[2]
```

```
Start date                  2017-10-02 06:37:10
End date                    2017-10-02 06:42:53
...
Name: 1, dtype: object
```

Loading datetimes with parse_dates

```
# Import W20529's rides in Q4 2017
rides = pd.read_csv('capital-onebike.csv',
                    parse_dates = ['Start date', 'End date'])
# Or:
rides['Start date'] = pd.to_datetime(rides['Start date'],
                                     format = "%Y-%m-%d %H:%M:%S")
```

5. Loading datetimes with parse_dates
If we want Pandas to treat these columns as datetimes, we can make use of the argument parse_dates in read_csv(), and set it to be a list of column names, passed as strings. Now Pandas will read these columns and convert them for us to datetimes. Pandas will try and be intelligent and figure out the format of your datetime strings. In the rare case that this doesn't work, you can use the to_datetime() method that lets you specify the format manually.

Loading datetimes with parse_dates

```
# Select Start date for row 2
rides['Start date'].iloc[2]
```

```
Timestamp('2017-10-02 06:37:10')
```

6. Loading datetimes with parse_dates
Now when we again ask for the Start date for row 2, we get back a Pandas Timestamp, which for essentially all purposes you can imagine is a Python Datetime object with a different name. They behave basically exactly the same.

Timezone-aware arithmetic

```
# Create a duration column
rides['Duration'] = rides['End date'] - rides['Start date']
# Print the first 5 rows
print(rides['Duration'].head(5))
```

```
0   00:03:01
1   02:07:02
2   00:05:43
3   00:21:18
4   00:21:17
Name: Duration, dtype: timedelta64[ns]
```

7. Timezone-aware arithmetic
Since our Start date and End date columns are now datetimes, we can deal with them the way we usually deal with datetimes. For example, we can create a new column, Duration, by subtracting Start date from End date. Because each of these columns are datetimes, when we subtract them we get timedeltas. If we print out the first 5 rows, we get that the first ride lasted for only 3 minutes and 1 second, the second ride lasted for 2 hours and 7 minutes, the third ride lasted for 5 minutes 43 seconds, and so on.

Loading datetimes with parse_dates

```
rides['Duration']\
  .dt.total_seconds()\
  .head(5)
```

```
0     181.0
1    7622.0
2     343.0
3    1278.0
4    1277.0
Name: Duration, dtype: float64
```

8. Loading datetimes with parse_dates

Pandas has two features worth noting here. Let's see an example of converting our Duration to seconds, and looking at the first 5 rows. First, Pandas code is often written in a "method chaining" style, where we call a method, and then another, and then another. For readability, it's common to break them up with a backslash and a linebreak at the end of each. Second, you can access all of the typical datetime methods within the namespace .dt. For example, we can convert our timedeltas into numbers with .dt.total_seconds(). Now when we look at the results, we see that we've got seconds instead of timedeltas. Our first ride lasted 181 seconds, our second ride 7622 seconds, and so on.

Reading date and time data in Pandas

WORKING WITH DATES AND TIMES IN PYTHON

Summarizing datetime data in Pandas

WORKING WITH DATES AND TIMES IN PYTHON

Max Shron
Data Scientist and Author

datacamp

Summarizing data in Pandas

```
# Average time out of the dock
rides['Duration'].mean()
```

```
Timedelta('0 days 00:19:38.931034')
```

```
# Total time out of the dock
rides['Duration'].sum()
```

```
Timedelta('3 days 22:58:10')
```

2. Summarizing data in Pandas
First things first, let's review some general principles for summarizing data in Pandas. You can call .mean(), .median(), .sum() and so on, on any column where it makes sense. For example, rides['Duration'].mean() returns that the average time the bike was out of the dock was 19 minutes and 38 seconds. We also can ask: how much is this column in total? By using the .sum() method, we can see that the bike was out of the dock for a total of 3 days, 22 hours, 58 minutes and 10 seconds during this time period.

Summarizing data in Pandas

```
# Percent of time out of the dock
rides['Duration'].sum() / timedelta(days=91)
```

```
0.04348417785917786
```

3. Summarizing data in Pandas
The output of Pandas operations mix perfectly well with the rest of Python. For example, if we divide this sum by 91 days (the number of days from October 1 to December 31), we see that the bike was out about 4.3% of the time, meaning about 96% of the time the bike was in the dock.

Summarizing data in Pandas

```
# Count how many time the bike started at each station
rides['Member type'].value_counts()
```

```
Member    236
Casual     54
Name: Member type, dtype: int64
```

4. Summarizing data in Pandas
For non-numeric columns, we have other ways of making summaries. The .value_counts() method tells us how many times a given value appears. In this case, we want to know how often the Member type is Member or Casual. 236 rides were from Members, and 54 were from Casual riders, who bought a ride at the bike kiosk without a membership. We can also divide by the total number of rides, using len(rides), and Pandas handles the division for us across our result. 81.4% of rides were from members, whereas 18.6% of rides were from casual riders.

```
# Percent of rides by member
rides['Member type'].value_counts() / len(rides)
```

```
Member    0.813793
Casual    0.186207
Name: Member type, dtype: float64
```

Summarizing datetime in Pandas

```
# Add duration (in seconds) column
rides['Duration seconds'] = rides['Duration'].dt.total_seconds()
# Average duration per member type
rides.groupby('Member type')['Duration seconds'].mean()
```

```
Member type
Casual    1994.666667
Member     992.279661
Name: Duration seconds, dtype: float64
```

5. Summarizing datetime in Pandas
Pandas has powerful ways to group rows together. First, we can group by values in any column, using the .groupby() method. .groupby() takes a column name and does all subsequent operations on each group. For example, we can groupby Member type, and ask for the mean duration in seconds for each member type. Rides from casual members last nearly twice as long on average.

Summarizing datetime in Pandas

```
# Average duration by month
rides.resample('M', on = 'Start date')['Duration seconds'].mean()
```

```
Start date
2017-10-31    1886.453704
2017-11-30     854.174757
2017-12-31     635.101266
Freq: M, Name: Duration seconds, dtype: float64
```

6. Summarizing datetime in Pandas
Second, we can also group by time, using the .resample() method. .resample() takes a unit of time (for example, 'M' for month), and a datetime column to group on, in this case 'Start date'. From this we can see that, in the month ending on October 31st, average rides were 1886 seconds, or about 30 minutes, whereas for the month ending December 31, average rides were 635 seconds, or closer to ten minutes.

Summarizing datetime in Pandas

```
# Size per group
rides.groupby('Member type').size()
```

```
Member type
Casual      54
Member     236
dtype: int64
```

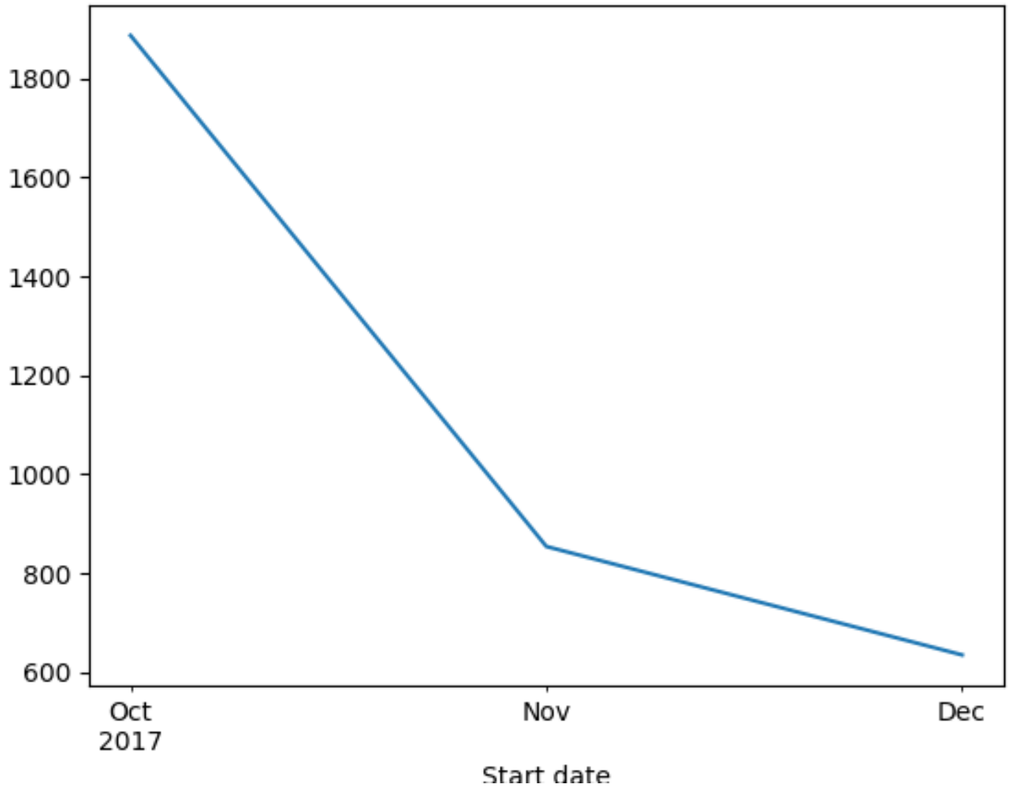
```
# First ride per group
rides.groupby('Member type').first()
```

```
             Duration     ...
Member type               ...
Casual       02:07:02     ...
Member       00:03:01     ...
```

7. Summarizing datetime in Pandas
There are also others methods which operate on groups. For example, we can call .size() to get the size of each group. Or we can call .first() to get the first row of each group.

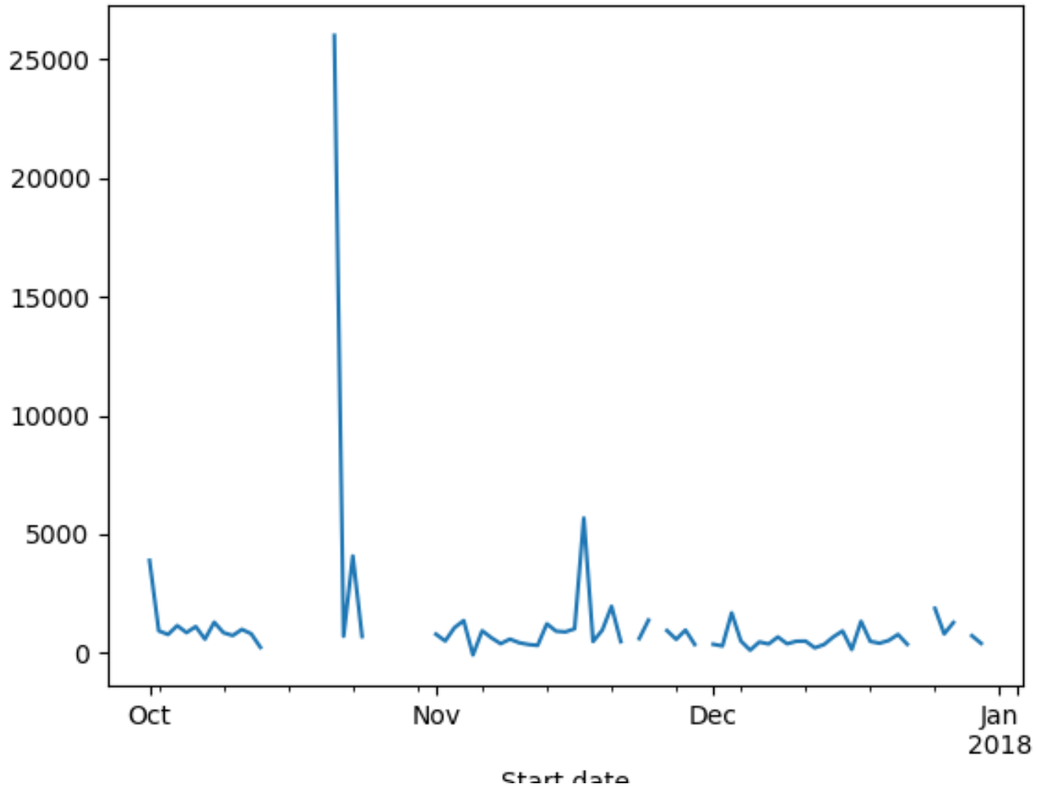
Summarizing datetime in Pandas

```
rides\
   .resample('M', on = 'Start date')\
   ['Duration seconds']\
   .mean()\
   .plot()
```



Summarizing datetime in Pandas

```
rides\
   .resample('D', on = 'Start date')\
   ['Duration seconds']\
   .mean()\
   .plot()
```



Summarizing datetime data in Pandas

WORKING WITH DATES AND TIMES IN PYTHON

Additional datetime methods in Pandas

WORKING WITH DATES AND TIMES IN PYTHON

Max Shron
Data Scientist & Author

datacamp

Timezones in Pandas

```
rides['Duration'].dt.total_seconds().min()
```

```
-3346.0
```

Timezones in Pandas

```
rides['Start date'].head(3)
```

```
0   2017-10-01 15:23:25
1   2017-10-01 15:42:57
2   2017-10-02 06:37:10
Name: Start date, dtype: datetime64[ns]
```

```
rides['Start date'].head(3)\
  .dt.tz_localize('America/New_York')
```

```
0   2017-10-01 15:23:25-04:00
1   2017-10-01 15:42:57-04:00
2   2017-10-02 06:37:10-04:00
Name: Start date, dtype: datetime64[ns, America/New_York]
```

3. Timezones in Pandas
The answer, as it was when we looked at this data set in standard Python, is Daylight Saving. Just like with standard Python, these datetime objects start off as timezone-naive. They're not tied to any absolute time with a UTC offset. Let's see the first three Start dates so we can see how they're displayed and check that there is no UTC offset. To start, we want those same three datetimes to be put into a timezone. The method for this in Pandas is .dt.tz_localize(). Now when we look at the localized datetimes, we can see that they have a UTC offset.

Timezones in Pandas

```
# Try to set a timezone...
rides['Start date'] = rides['Start date']\
  .dt.tz_localize('America/New_York')
```

```
AmbiguousTimeError: Cannot infer dst time from '2017-11-05 01:56:50',
try using the 'ambiguous' argument
```

```
# Handle ambiguous datetimes
rides['Start date'] = rides['Start date']\
  .dt.tz_localize('America/New_York', ambiguous='NaT')

rides['End date'] = rides['End date']\
  .dt.tz_localize('America/New_York', ambiguous='NaT')
```

4. Timezones in Pandas

However, if we try to convert our entire Start date column to the America/New_York timezone, Pandas will throw an AmbiguousTimeError. As expected, we have one datetime that occurs during the Daylight Saving shift. Following the advice of the error message, we can set the ambiguous argument in the .dt.tz_localize() method. By default, it raises an error, as we saw above. We also can pass the string 'NaT', which says that if the converter gets confused, it should set the bad result as Not a Time. Pandas is smart enough to skip over NaTs when it sees them, so our .min() and other methods will just ignore this one row.

Timezones in Pandas

```
# Re-calculate duration, ignoring bad row
rides['Duration'] = rides['Start date'] - rides['End date']
# Find the minimum again
rides['Duration'].dt.total_seconds().min()
```

```
116.0
```

Timezones in Pandas

```
# Look at problematic row
rides.iloc[129]
```

```
Duration                          NaT
Start date                        NaT
End date                          NaT
Start station          6th & H St NE
End station            3rd & M St NE
Bike number                   W20529
Member type                   Member
Name: 129, dtype: object
```

4. Timezones in Pandas
However, if we try to convert our entire Start date column to the America/New_York timezone, Pandas will throw an AmbiguousTimeError. As expected, we have one datetime that occurs during the Daylight Saving shift. Following the advice of the error message, we can set the ambiguous argument in the .dt.tz_localize() method. By default, it raises an error, as we saw above. We also can pass the string 'NaT', which says that if the converter gets confused, it should set the bad result as Not a Time. Pandas is smart enough to skip over NaTs when it sees them, so our .min() and other methods will just ignore this one row.

Other datetime operations in Pandas

```
# Year of first three rows
rides['Start date']\
  .head(3)\
  .dt.year
```

```
0    2017
1    2017
2    2017
Name: Start date, dtype: int64
```

```
# See weekdays for first three rides
rides['Start date']\
  .head(3)\
  .dt.weekday_name
```

```
0    Sunday
1    Sunday
2    Monday
Name: Start date, dtype: object
```

7. Other datetime operations in Pandas
There are other datetime operations you should know about too. The simplest are ones you're already familiar with: .year, .month, and so on. In Pandas, these are accessed with .dt.year, .dt.month, etc. Here, for example, is the year of the first three rows. There are other useful properties that Pandas gives you, some of which are not available in standard Python. For example, the attribute .dt.weekday_name gives a Series which is the name of the weekday for each element in a datetime series. This can be used in .groupby() calls too.

Other parts of Pandas

```
# Shift the indexes forward one, padding with NaT
rides['End date'].shift(1).head(3)
```

```
0                          NaT
1   2017-10-01 15:26:26-04:00
2   2017-10-01 17:49:59-04:00
Name: End date, dtype: datetime64[ns, America/New_York]
```

8. Other parts of Pandas
Pandas also lets you shift rows up or down with the .shift() method. Here we've shifted the rides one row forward so that our zeroth row is now NaT, and our first row has the same value that our zeroth row had before. This is useful if you want to, for example, line up the end times of each row with the start time of the next one.

Additional datetime methods in Pandas

WORKING WITH DATES AND TIMES IN PYTHON

Wrap-up

WORKING WITH DATES AND TIMES IN PYTHON

Max Shron
Data Scientist and Author

Recap: Dates and Calendars

- The `date()` class takes a year, month, and day as arguments
- A `date` object has accessors like `.year`, and also methods like `.weekday()`
- `date` objects can be compared like numbers, using `min()`, `max()`, and `sort()`
- You can subtract one `date` from another to get a `timedelta`
- To turn `date` objects into strings, use the `.isoformat()` or `.strftime()` methods

Recap: Combining Dates and Times

- The `datetime()` class takes all the arguments of `date()`, plus an hour, minute, second, and microsecond
- All of the additional arguments are optional; otherwise, they're set to zero by default
- You can replace any value in a `datetime` with the `.replace()` method
- Convert a `timedelta` into an integer with its `.total_seconds()` method
- Turn strings into dates with `.strptime()` and dates into strings with `.strftime()`

Recap: Timezones and Daylight Saving

- A `datetime` is "timezone aware" when it has its `tzinfo` set. Otherwise it is "timezone naive"
- Setting a timezone tells a `datetime` how to align itself to UTC, the universal time standard
- Use the `.replace()` method to change the timezone of a `datetime`, leaving the date and time the same
- Use the `.astimezone()` method to shift the date and time to match the new timezone
- `dateutil.tz` provides a comprehensive, updated timezone database

Recap: Easy and Powerful Timestamps in Pandas

- When reading a csv, set the `parse_dates` argument to be the list of columns which should be parsed as datetimes
- If setting `parse_dates` doesn't work, use the `pd.to_datetime()` function
- Grouping rows with `.groupby()` lets you calculate aggregates per group. For example, `.first()`, `.min()` or `.mean()`
- `.resample()` groups rows on the basis of a `datetime` column, by year, month, day, and so on
- Use `.tz_localize()` to set a timezone, keeping the date and time the same
- Use `.tz_convert()` to change the date and time to match a new timezone

Congratulations!

WORKING WITH DATES AND TIMES IN PYTHON