**Time zone database**

Now that you understand how UTC offsets work, it's time to talk about how you use timezones in practice.

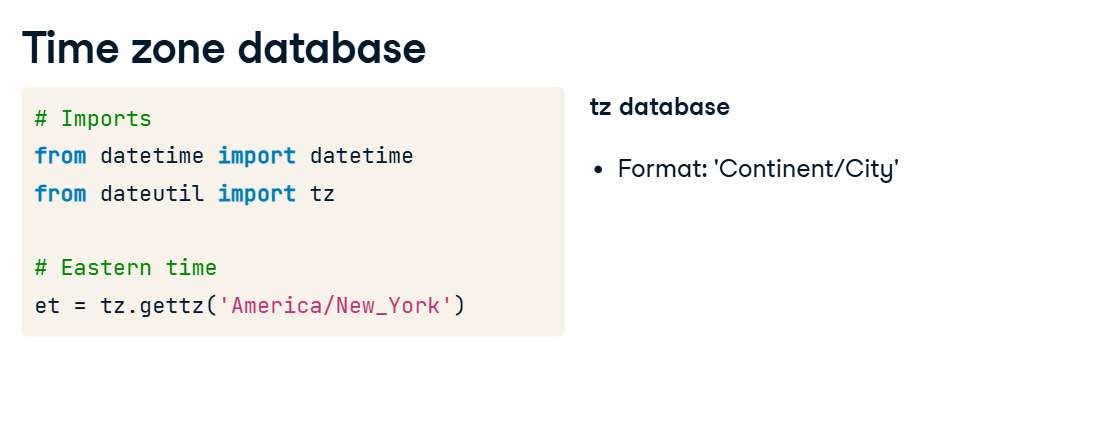
**Time zone database**

This is a picture of all of the different time zones in the world, as of 2017. They cut across countries, and within countries, and sometimes one is even totally surrounded by another one. How could you possibly know all of these when you need to align your data to UTC? Do you need to look up the offset for each one in some big spreadsheet somewhere? Can't a computer help with this?

Thankfully, yes. There is a database called tz, updated 3-4 times a year as timezone rules change. This database is used by computer programs across many programming languages. Because timezone information changes so quickly, it doesn't make sense to bundle it directly into Python. Instead, you will use a package called dateutil.

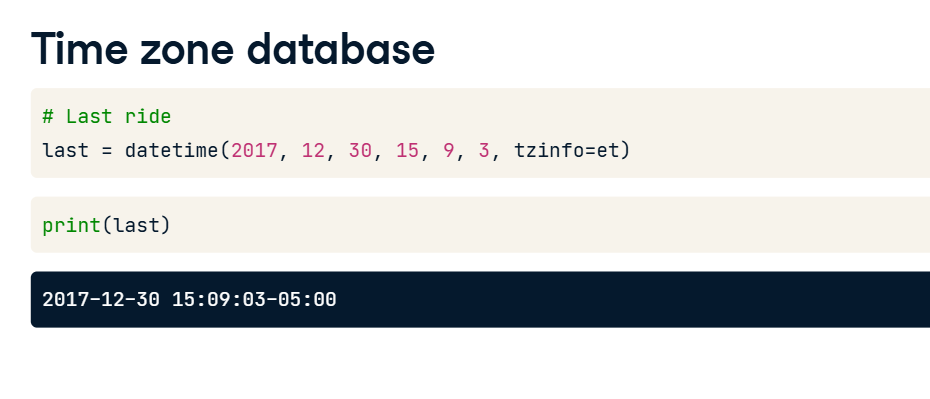
**Time zone database**

Let's start by making a timezone object that corresponds to the eastern United States, where our bicycle data comes from. Within tz, time zones are defined first by the continent they are on, and then by the nearest major city. For example, the time zone used on the eastern seaboard of the United States is 'America/New York'. We fetch this timezone by calling tz-dot-gettz(), and passing 'America/New York' as the string.



Here are a few more examples: 'America/Mexico\_City'. 'Europe/London'. 'Africa/Accra'.

Let's look at our last ride again. Instead of specifying the UTC offset yourself, you pass the timezone you got from tz. Look at the result, and you can see that it's got the right UTC offset.



**Time zone database**

Even more excitingly, this same object will adjust the UTC offset depending on the date and time. If we call datetime() with the time of our first ride, and pass in the same timezone info, we see that it gives us a different UTC offset. We will discuss daylight savings time in the next lesson, but suffice to say that in some places the clocks change twice a year. Instead of having to look up when these things change, we just ask the timezone database to know for us. tz includes rules for UTC offsets going all the way back to the late 1960s, and sometimes earlier. If you have data stretching over a long period of time, and you really care about getting the exact hours and minutes correct, you can use tz to put all of your date and timestamps on to a common scale.

**1. Starting Daylight Saving Time**

Some places change their clocks twice a year to create longer summer evenings. This practice is called daylight saving time, but it would better be called daylight shifting time. In some countries it is called "summer time". Dealing with daylight saving time can be one of the most fiendish challenges in dealing with dates and times. To keep things simple, let's start with the situation where the clocks move forward in the spring. In the next lesson, we'll discuss handling the opposite case, when the clocks move back in the fall.

**2. Start of Daylight Saving Time**

Let's look at an example. On March 12, 2017, in Washington, DC, the clock jumped straight from 1:59 am to 3 am. The clock "springs forward". It never officially struck 2 am anywhere on the East Coast of the United States that day.

**3. Start of Daylight Saving Time**

Just like before, to make our clock in Washington, DC comparable to clocks in other places, we need to represent it with a UTC offset. Only now the UTC offset is going to change. On this date, at 1 AM in Washington, DC, we were in Eastern Standard Time. It was 6 AM UTC, a five-hour difference. At 3 AM in Washington, DC, we were in Eastern Daylight Time. It was 7 AM UTC, a four-hour difference.

**4. Start of Daylight Saving Time**

Let's see the same thing in code. To be as clear as possible, let's create the UTC offsets by hand for now instead of using dateutil-dot-tz. We start by creating a datetime object, spring\_ahead\_159am, for March 12th, at 1:59:59, without any timezone information. We print the results out with isoformat() to check that we have the time right, and we make another object for spring\_ahead\_3am. We subtract the two datetime objects and ask how much time has elapsed by calling total\_seconds(). As expected, they're an hour and one second apart.

**5. Start of Daylight Saving Time**

As before, to fix problems with comparing datetimes we start by creating timezone objects. We define Eastern Standard Time, or EST, using the timezone constructor. We set the offset to -5 hours. Similarly, we define Eastern Daylight Time, or EDT, with an offset of -4 hours.

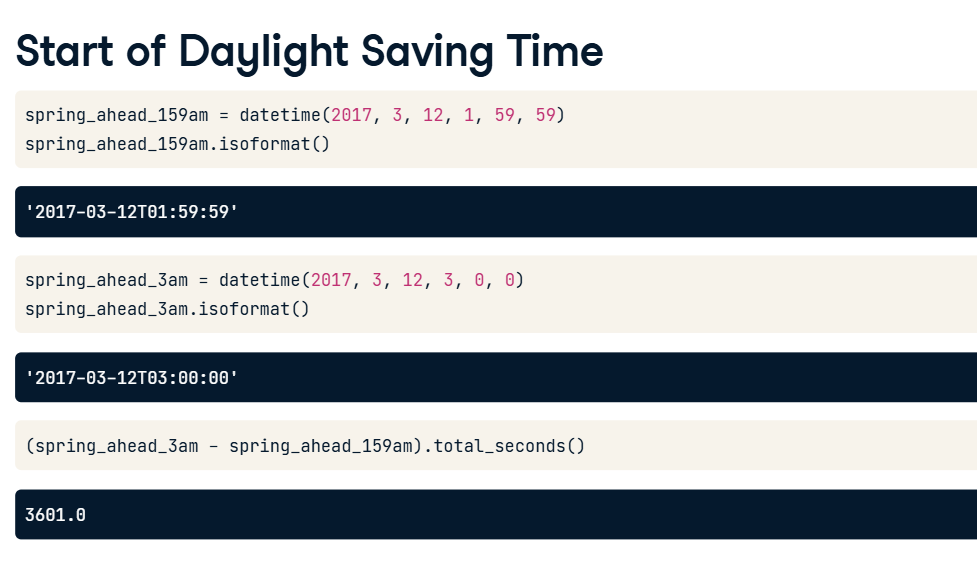
**6. Start of Daylight Saving Time**

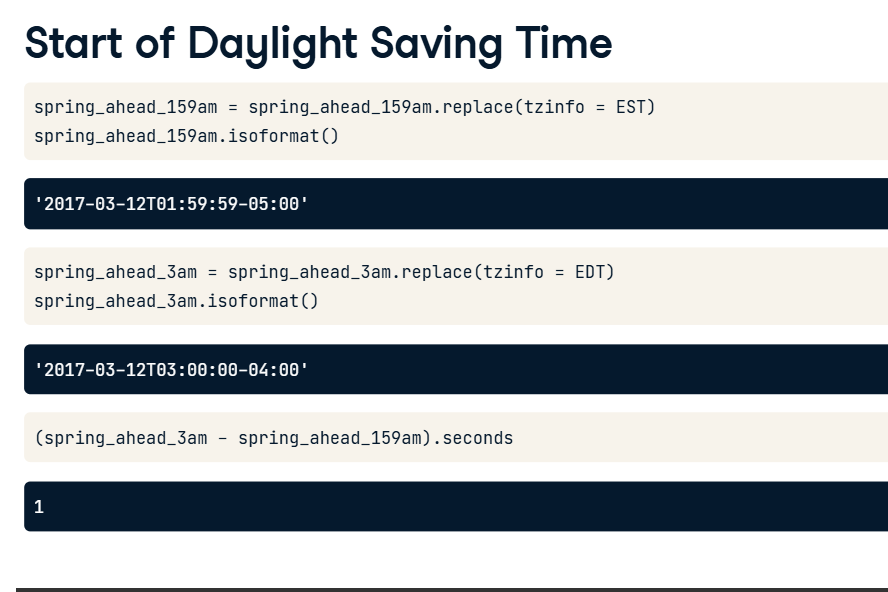
We assign our first timestamp, at 1:59 am to be in EST. When we call isoformat(), we see it has the correct offset. We assign our second timestamp, at 3:00 am, to be in EDT, and again check the output with isoformat(). When we subtract the two datetime objects, we see correctly that one second has elapsed. Putting things in terms of UTC once again allowed us to make proper comparisons.

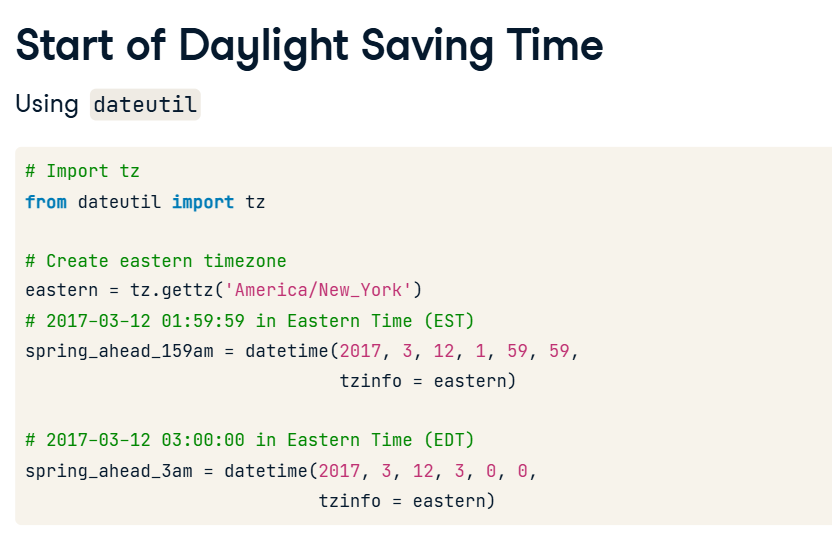
**7. Start of Daylight Saving Time**

But how do we know when the cutoff is without looking it up ourselves? dateutil to the rescue again. Just like before when it saved us from having to define timezones by hand, dateutil saves us from having to know daylight savings rules. We create a timezone object by calling tz-dot-gettz() and pass our timezone description string. Recall that since Washington, DC is in the America/New\_York time zone, that's what we use. Once again we create a datetime corresponding to 1:59 am on the day that the east coast of the US springs forward. This time though, we set the tzinfo to eastern time. Similarly, we create a datetime set to 3 am on March 12th, and when we set tzinfo to be eastern time, dateutil figures out for us that it should be in EDT.









**Reading date and time data in Pandas**

In this chapter, you will use the Pandas library to work with dates and times. You should have encountered Pandas before, but now we will add datetimes to the mix.

**A simple Pandas example**

To start with, let's load data with Pandas. First, we import pandas, and as is customary we use the alias pd. Our data is in a csv file, so we load it with the read\_csv() function. pd-dot-read\_csv() has one required argument, the name of the file to load, which in this case is capital-onebike-dot-csv. We save the result to the variable rides. Let's print the first three rows to see what we've got.

**A simple Pandas example**

Note that the index, listed all the way to the left, starts with zero. Because the table is too wide, it wraps around. Each of these three rows has a start date, an end date, a start station, and end station, the bike number, and whether the ride was from someone who is a member or someone who walked up to the kiosk and bought a ride on the spot.

**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 dot-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.

**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. For more details, see the Pandas documentation.

**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**

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**

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 -dot-dt. For example, we can convert our timedeltas into numbers with dot-dt-dot-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**

In this lesson, we discussed loading data in Pandas, and handling basic datetime elements. We talked about using slashes to continue lines, and selecting subsets of rows. Time to practice!

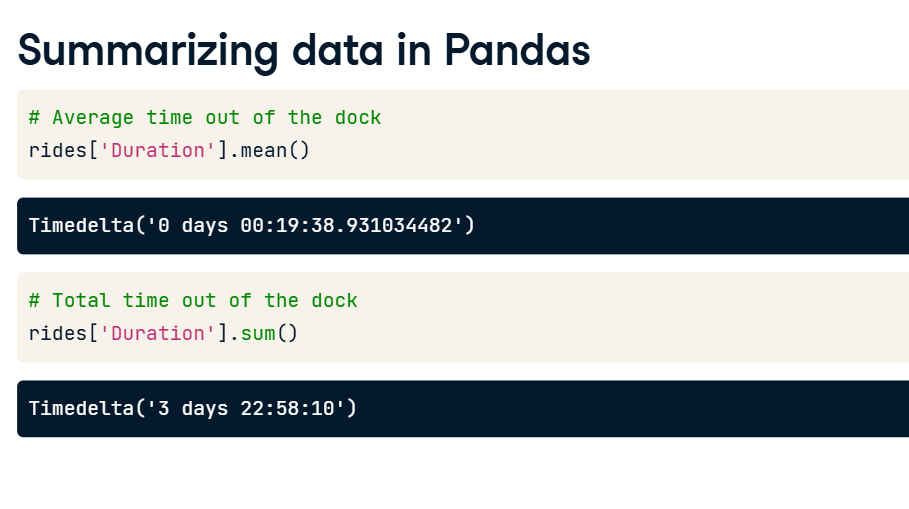
**Summarizing datetime data in Pandas**

In this lesson, we will discuss how to summarize Pandas tables, especially when we have datetime columns. One note: Pandas continues to evolve quickly. Many of the techniques in this chapter don't work on versions of Pandas more than a few years old. If anything breaks on your personal computer, make sure you're using at least Pandas version 0-point-23.

**2. Summarizing data in Pandas**

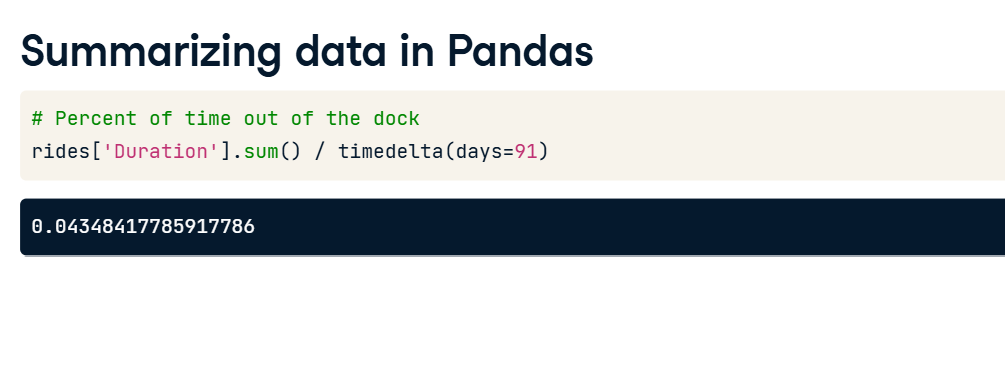
00:28 - 01:10

First things first, let's review some general principles for summarizing data in Pandas. You can call dot-mean(), dot-median(), dot-sum() and so on, on any column where it makes sense. For example, rides['Duration']-dot-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 dot-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**

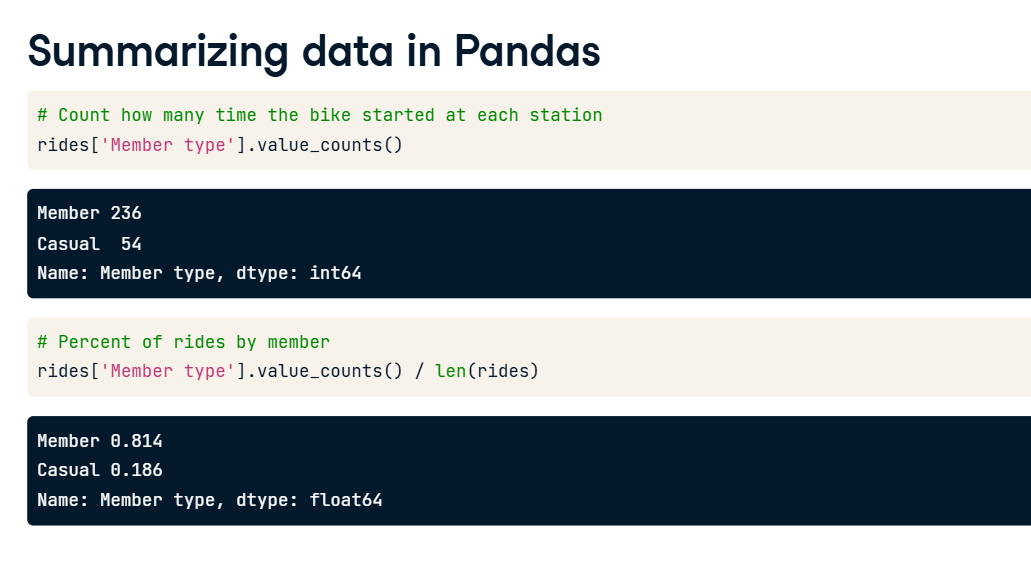
The output of Pandas operations mix perfectly well with the rest of Python. For example, if we divide this sum by 91 days (the number of days from October 1 to December 31), we see that the bike was out about 4.3% of the time, meaning about 96% of the time the bike was in the dock.



**4. Summarizing data in Pandas**

01:32 - 02:18

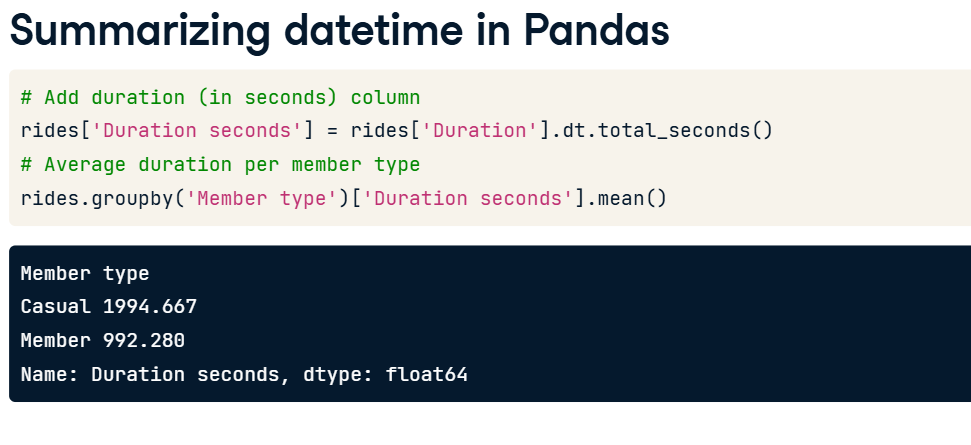
For non-numeric columns, we have other ways of making summaries. The dot-value\_counts() method tells us how many times a given value appears. In this case, we want to know how often the Member type is Member or Casual. 236 rides were from Members, and 54 were from Casual riders, who bought a ride at 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-point-4% of rides were from members, whereas 18-point-6% of rides were from casual riders.



**5. Summarizing datetime in Pandas**

02:18 - 02:58

To make this next section easy, let's make a column called 'Duration seconds', which will be the original column 'Duration' converted to seconds. Pandas has powerful ways to group rows together. First, we can group by values in any column, using the dot-groupby() method. dot-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.



**6. Summarizing datetime in Pandas**

02:58 - 03:33

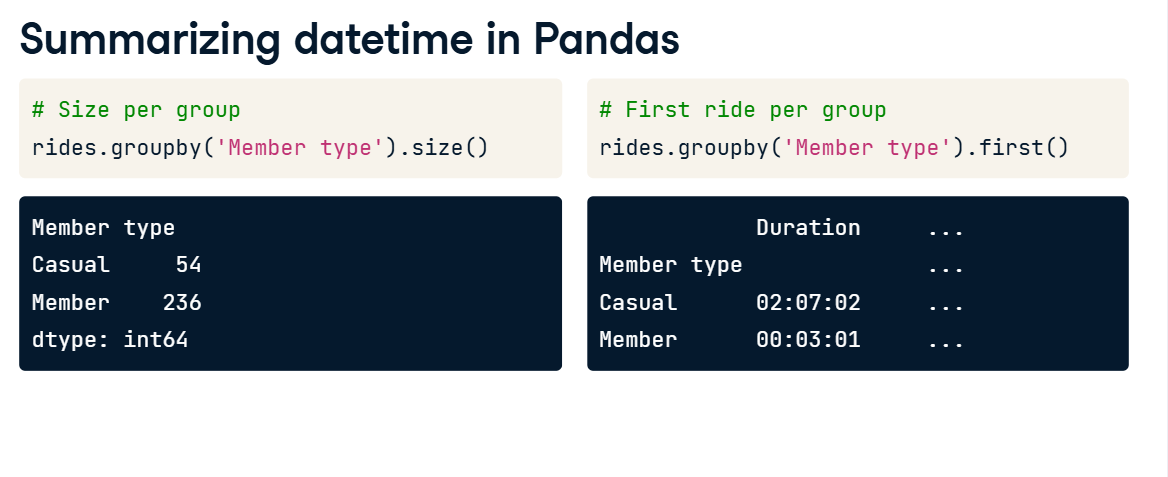
Second, we can also group by time, using the dot-resample() method. dot-resample() takes a unit of time (for example, 'M' for month), and a datetime column to group on, in this case 'Start date'. From this we can see that, in the month ending on October 31st, average rides were 1886 seconds, or about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



**7. Summarizing datetime in Pandas**

03:33 - 03:48

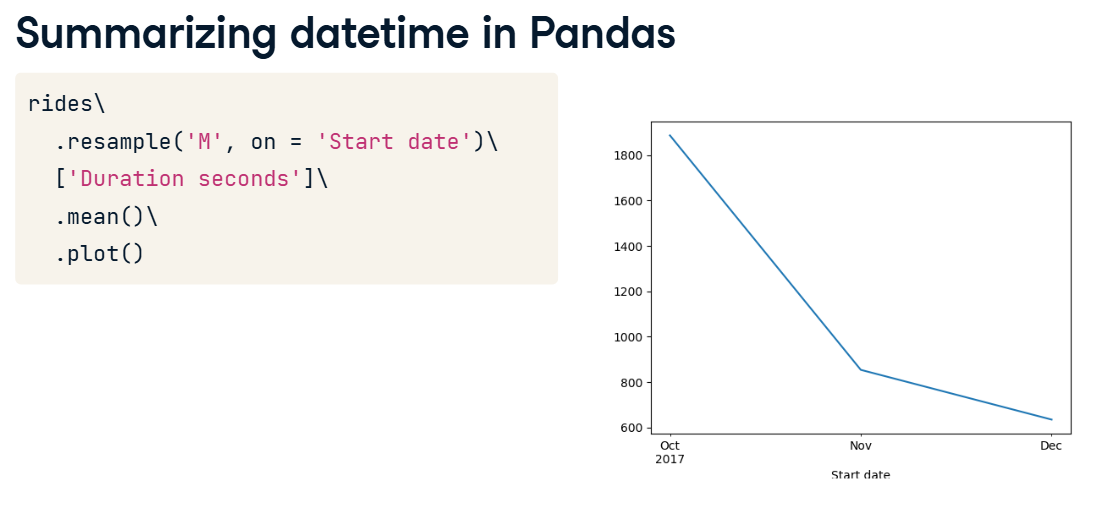
There are also others methods which operate on groups. For example, we can call dot-size() to get the size of each group. Or we can call dot-first() to get the first row of each group.



**8. Summarizing datetime in Pandas**

03:48 - 04:05

Pandas also makes it easy to plot results. Just add the dot-plot() method at the end of your call and it will pass the results to the Python plotting library Matplotlib. It will usually have sensible defaults, though if you want to change things further you can.



**9. Summarizing datetime in Pandas**

04:05 - 04:47

We can also change the resampling rate from 'M' for months to 'D' for days, and plot again. Now we can see that there is at least one big outlier skewing our data: some ride in the middle of October was 25000 seconds long, or nearly 7 hours. We identified this ride in an earlier chapter as possibly a bike repair. Now we can see that it happened after many days with zero rides, which lends strength to that idea. If the bike was broken and sitting in the dock for awhile, eventually it would have been removed for repairs, then returned.

**10. Summarizing datetime data in Pandas**

In this lesson, we discussed how to use basic Pandas operations, such as dot-mean(), dot-median() and dot-sum(), and also dot-groupby() and dot-resample() to combine our rows into different groups. Time to practice what you've learned!

**1. Additional datetime methods in Pandas**

00:00 - 00:17

In this final lesson, we will cover some additional Pandas methods for working with dates and times. By the end of this lesson, you will understand how to handle timezones in Pandas, as well as other common datetime operations.

**2. Timezones in Pandas**

00:17 - 00:45

First, a reminder of the importance of timezones. If we ask Pandas to tell us the smallest ride duration in seconds, using the dt-dot-total\_seconds() method and then the dot-min() method, we get -3346 seconds, or -55 minutes. Yikes! Something is wrong, since our ride durations shouldn't ever be negative.

**3. Timezones in Pandas**

00:45 - 01:36

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-dot-tz\_localize(). Now when we look at the localized datetimes, we can see that they have a UTC offset.



**4. Timezones in Pandas**

01:36 - 02:24

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-dot-tz\_localize() method. By default, it raises an error, as we saw before. 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 dot-min() and other methods will just ignore this one row.

**5. Timezones in Pandas**

02:24 - 02:45

Now that we've fixed the timezones, we should recalculate our durations, in case any rides had been across Daylight Saving boundaries. This time, when we take Durations, convert it to seconds, and take the minimum, we get a much more sensible 116-point-0 seconds, or about two minutes.

**6. Timezones in Pandas**

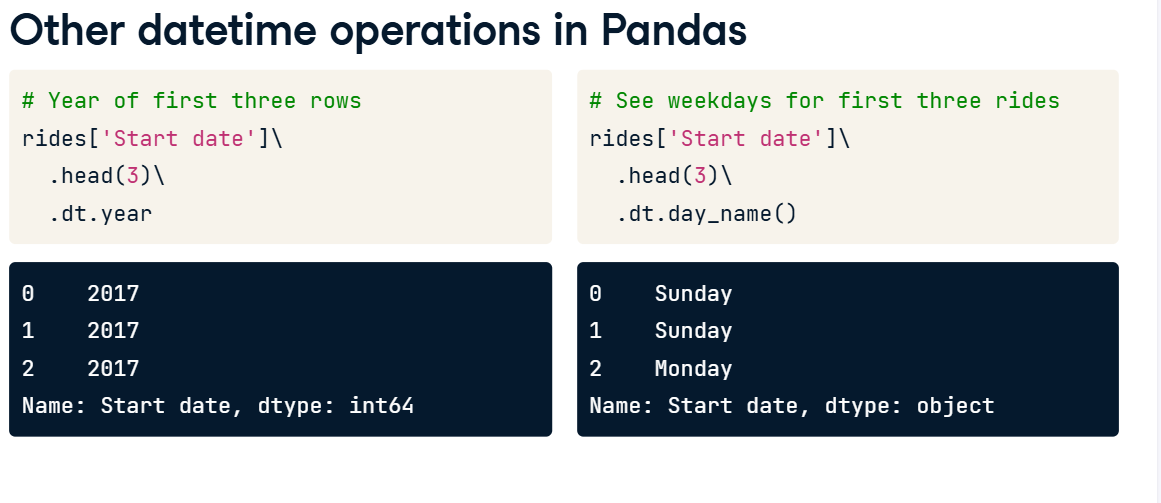
02:45 - 03:03

Just to know what we're looking at, let's pull up our problematic row. Here, both the start and end time were ambiguous, so they've been set to NaT. As a result, our Duration, since it's the difference of two undefined times, is also NaT.

**7. Other datetime operations in Pandas**

03:03 - 04:00

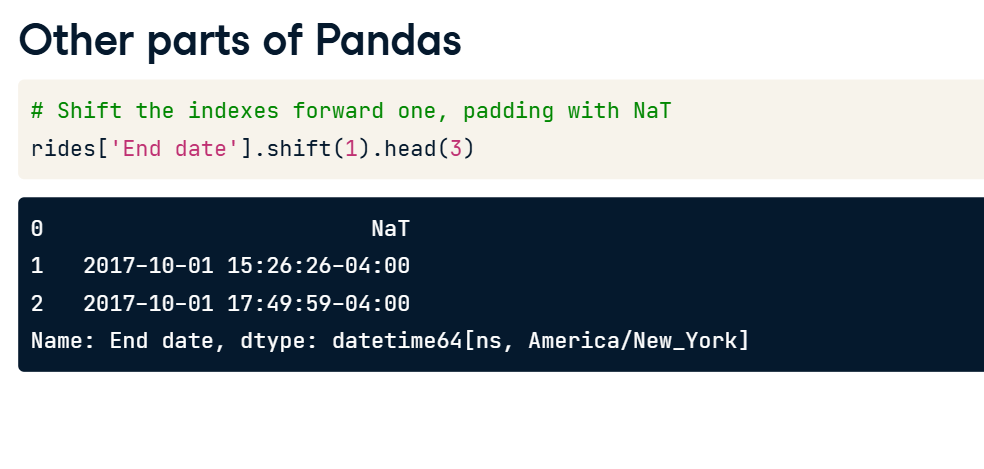
There are other datetime operations you should know about too. The simplest are ones you're already familiar with: .year, .month, and so on. In Pandas, these are accessed with dt-dot-year, dt-dot-month, etc. For example, here is the year of the first three rows. There are other useful things that Pandas gives you, some of which are not available in standard Python. For example, the method dt-dot-day\_name() gives you the day of the week for each element in a datetime Series. You can even specify if you want weekday names in a language other than English. These results can be aggregated with dot-groupby() call, to summarize data by year, month, day of the week, and so on.



**8. Other parts of Pandas**

04:00 - 04:35

Pandas also lets you shift rows up or down with the dot-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. Now you can answer questions about how each ride compares to the previous one! You'll cover this in an exercise shortly.



**9. Additional datetime methods in Pandas**

04:35 - 04:52

In this lesson, we looked at additional methods in Pandas that are relevant to working with datetimes. Hopefully, this gave you a good taste of all the things Pandas is capable of! Time to try them out in the exercises.

**2. Recap: Dates and Calendars**

00:08 - 00:46

In the first chapter of this course, we covered dates in Python. 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.

**3. Recap: Combining Dates and Times**

00:46 - 01:29

In the second chapter of this course, we covered datetimes. 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().

**4. Recap: Timezones and Daylight Saving**

01:29 - 02:16

In the third chapter of this course, we covered 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-dot-tz provides a comprehensive, updated timezone database.

**5. Recap: Easy and Powerful Timestamps in Pandas**

02:16 - 03:13

In the fourth and final chapter of this course, we covered using Pandas for handling dates and times. 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-dot-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.

**6. Congratulations!**

03:13 - 03:35

At this point, you have all of the knowledge you need to effectively use dates and times in Python. Even better, you've had a chance to practice! If you want to get really good with dates, why not get some more practice? Google for "interesting data sets", find one you like that has dates in it, pull it down, and get going!