10 Dates and times

Programming languages' core data structures reflect the types of information we work with on a regular basis. It makes sense that we'll have numbers, because we use numbers a lot. We use lots of text, so strings make sense, as well. And of course we need collections of various sorts, so every language provides some of those—in the case of Python, we have lists, tuples, dictionaries, and sets, for starters.

Modern programming languages also support another type of data, one that we (as people) use on a regular basis but that wasn't part of the programming canon when I started my career: dates and times. It seems obvious in retrospect that dates and times, which are such essential parts of our lives, should be a main part of our programming

languages. But it turns out that dealing with dates and times is hard, with all sorts of tricky problems—from leap years, to time zones, to the odd data structures we need to computerize a calendar that wasn't exactly designed with computers in mind.

Both the Python language and pandas handle time data with two different data structures. The first is a *timestamp*, also known as a *datetime* in many languages and systems. A timestamp refers to a specific point in time that you can point to using a calendar. A timestamp happens once and only once—when you were born, when your plane will take off, when you and your date will meet at a restaurant, or when the meeting was scheduled to end. You can describe a timestamp with a particular year, month, day, hour, minute, and second.

A second, complementary data type is the *timedelta*, known in some systems as an *interval*. A timedelta represents a time span—the distance between two timestamp objects. So a meeting's scheduled start and

end can be represented as timestamps, but the time the meeting takes is a timedelta (figure 10.1).

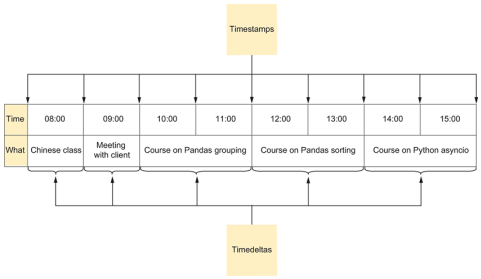


Figure 10.1 My schedule, illustrated with timestamps and timedeltas

Not surprisingly, lots of the data we want to analyze contains time and date information. And thus it's good to know that pandas can handle dates and times flexibly. We can read data in from files, turning columns into timestamps. We can also convert existing values—both individual values and series objects—into timestamps. We can perform calculations with timedeltas and perform comparisons with them.

But pandas goes further than that, allowing us to use date and time information in indexes. This makes it easier to search for data that took place during specific periods. Even better, we can perform

resampling, which is most easily described as *grouping by time periods*.

This chapter's exercises all take advantage of these capabilities in pandas to explore information that has to do with dates and times. Along the way, you'll get experience working with a variety of date formats and input types, as well as producing reports based on those types.

Table 10.1 What you need to

know

Concept	What is it?	Example	To learn more
`pd.to_datetime`	If passed a series of strings, returns a series of `Timestamp` objects	`pd.to_datetime(s['when'])`	**http://mng.bz/6D2D**
`pd.to_timedelta`	If passed a series of strings, returns a series of `Timedelta` objects	`pd.to_timedelta(s['how_long'])`	**http://mng.bz/o1Er**
`pd.read_csv`	Returns a new data frame based on CSV input	`df = pd.read_csv('myfile.csv')`	**http://mng.bz/nW8g**
`time.strftime`	Produces a string based on a time value	`time.strftime(a_time, a_format)`	**http://mng.bz/vn5J**
`time.strptime`	Parses a string into a time object	`time.strptime(time_string)`	**http://mng.bz/4D2a**

`df.to_csv`	Writes a CSV file based on a data frame	`df.to_csv ('mydata.csv')`	http://mng.bz/QPNw
`df.resample`	Performs a time-based `groupby` operation on a specified period of time	`df.resample('1M')`	http://mng.bz/XN6G
`s.diff`	Returns a new series with the same index as `s` but whose values indicate the difference between that value and the previous value	`s.diff()`	http://mng.bz/yQnG
`s.pct_change`	Returns a new series with the same index as `s` but whose	`s.pct_change()`	http://mng.bz/MBj7

values indicate the *percentage* difference between that value and the previous value

Creating datetime and timedelta objects

As we've repeatedly seen, pandas largely avoids built-in Python data structures in favor of its own types or those defined by NumPy. This is also the case when it comes to dates and times: to represent a specific point in time, we use the `Timestamp` class instead of either the `datetime.datetime` class that comes with Python or the `np.datetime64` class that comes with NumPy.

The standard way to create `Timestamp` objects is with the module-level function `to_datetime`, which takes a variety of argument types. If passed a single argument, it returns one `Timestamp`. For example, we can get the

current date and time by passing it the string `'now'`:

```
pd.to_datetime('now')
```

It's far more common and useful to call `pd.to_datetime` on an existing series of strings containing date and time information. For example:

```
s = Series(['1970-07-14', '1972-03-01', '2000-12-16',
    '2002-12-17', '2005-10-31'])
pd.to_datetime(s)
```

This code returns a new series:

```
0   1970-07-14
1   1972-03-01
2   2000-12-16
3   2002-12-17
4   2005-10-31
dtype: datetime64[ns]
```

Don't be confused by the indication that the `dtype` is `datetime64`, a type from NumPy; the values are all of type `Timestamp`, a pandas type.

In this example, the strings we feed to `to_datetime` are unambiguous and easy to

parse. But what if we have slightly different strings, using month names instead of numbers?

```
s = Series(['1970-Jul-14', '1972-Mar-01', '2000-Dec-16',
    '2002-Dec-17', '2005-Oct-31'])
pd.to_datetime(s)
```

Actually, this works fine: that's because `pd.to_datetime` is fairly smart and flexible and can parse a number of different date formats. So this format works, as does this one:

```
s = Series(['14-Jul-1970', '01-Mar-1972', '16-Dec-2000',
    '17-Dec-2002', '31-Oct-2005'])
pd.to_datetime(s)
```

But what if we pass dates that are more ambiguous? For example, what if the months are all numbers?

```
s = Series(['14-07-1970', '01-03-1972', '16-12-2000',
    '17-12-2002', '31-10-2005'])
pd.to_datetime(s)
```

Once again, it works fine. However, sometimes dates are less obvious and human culture and tradition play a

role. For example, take the following:

```
s = Series(['01/03/1972', '05/12/1995'])
pd.to_datetime(s)
```

Should pandas interpret these dates as March 1 or January 3, and as December 5 or May 12? By default, ambiguous date formats are assumed to have the month

first, as in the United States. However, you can override that by passing `dayfirst=False` to `pd.to_datetime`:

```
s = Series(['01/03/1972', '05/12/1995'])
pd.to_datetime(s, dayfirst=False)
```

These examples have only included dates, but we can include time information, as well:

```
s = Series(['1970-07-14 8:00', '1972-03-01 10:00 pm',
    '2000-12-16 12:15:28', '2002-12-17 18:17', '2005-10-31 23:51'])
pd.to_datetime(s)
```

This code returns

```
0   1970-07-14 08:00:00
1   1972-03-01 22:00:00
```

```
2   2000-12-16 12:15:28
3   2002-12-17 18:17:00
4   2005-10-31 23:51:00
dtype: datetime64[ns]
```

Notice that we sometimes include seconds and in one case indicate a.m./p.m. rather than using a 24-hour clock. Pandas tries hard to understand all these formats and interpret them as well as possible.

What if we have several series representing the year, month, and date? We can use `pd.to_datetime` to get a new `Timestamp` series based on those inputs. This is especially useful if we're trying to create a `Timestamp` column from a data frame:

```
df = DataFrame([s.split('-')
                for s in ['14-07-1970', '01-03-1971',
                   '16-12-2000', '17-12-2002',
                   '31-10-2005']],
           columns='day month year'.split())
pd.to_datetime(df[['year', 'month', 'day']])
```

This code results in

```
0   1970-07-14
1   1971-03-01
2   2000-12-16
3   2002-12-17
```

```
4   2005-10-31
dtype: datetime64[ns]
```

All this is fine, but it ignores a common use case: loading a CSV file in which one or more columns are datetime information. How can we ensure that these columns are interpreted as `Timestamp` data and not as strings? We need to tell pandas to do this, using the `parse_dates` parameter in the `read_csv` function. We can pass a list of columns, either as names (strings) or as integers (indexes). For example:

```
pd.read_csv(filename,
            parse_dates=['birthday', 'anniversary'])
```

We can pass various parameters to influence the parsing process. One of them is `dayfirst`, which works as we saw earlier to indicate that the dates being read start with days (as in Europe) rather than with months (as in the United States).

Once we have a `Timestamp` series, we can use the `dt` accessor to retrieve different

parts of each object. For example:

```
s.dt.month          # month number
s.dt.month_name     # month name
s.dt.hour           # hour
s.dt.day_of_week    # day of week
s.dt.is_leap_year   # is it a leap year?
```

Some of these attributes return numbers, and others return boolean values. You can retrieve the full list of attributes via the `dt` accessor at **http://mng.bz/j1wa**.

Finally, I mentioned at the start of this chapter that when we work with dates and times, we need two distinct data types. We've spent some time looking at the first one: timestamps. But what about timedeltas, also known as intervals? We can generally say

```
datetime - datetime = interval
datetime + interval = datetime
datetime - interval = datetime
```

In other words, given two datetime objects, we can get an interval object representing the time between them. For example,

given a birth date and a death date, we can calculate the length of someone's life. And given a datetime and an interval, we can get the datetime on the other side of that interval. For example, given a meeting start time and its length, we can find out when it ends—or similarly, if given a meeting end time and its length, we can calculate when it started.

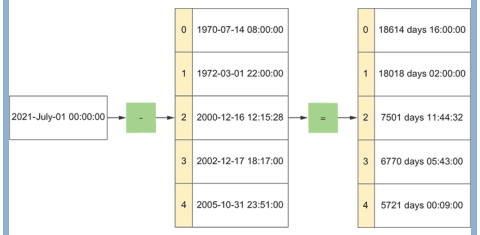
Pandas allows us to perform precisely this type of calculation. For example, if we have two timestamp series, subtracting one from the other gives us a `timedelta` series. For example:

```
s = Series(['1970-07-14 8:00', '1972-03-01 10:00 pm',
            '2000-12-16 12:15:28', '2002-12-17 18:17',
            '2005-10-31 23:51'])
s = pd.to_datetime(s)
pd.to_datetime('2021-July-01') - s
```

The subtraction operation is broadcast to every element in `s`, returning a series of `timedelta64` objects:

```
0   18614 days 16:00:00
1   18018 days 02:00:00
2    7501 days 11:44:32
```

```
3    6770 days 05:43:00
4    5721 days 00:09:00
dtype: timedelta64[ns]
```



Subtracting a timestamp from a timestamp gives us a timedelta.

To create a `timedelta` object or series, we can also call `pd.to_timedelta` much as we can call `pd.to_timestamp`. The function's argument is typically a string or series of strings, each describing a time span, such as `'1 hour'` or `'2 days'`.

The pieces of a `timedelta` can be retrieved using the `components` attribute. For example:

```
pd.to_timedelta('2 days 3:20:10').components
```

This returns

```
Components(days=2, hours=3, minutes=20, seconds=10,
           milliseconds=0, microseconds=0, nanoseconds=0)
```

If we have a timedelta object, we can use the `days`, `seconds`, `microseconds`, and `nanoseconds` attributes to retrieve those calculations.

Now that you've seen how you can create and retrieve from `timestamp` and `timedelta` objects, you're all set to work through the exercises in this chapter, which use these skills to answer questions about a number of data sets.

xercise 39 • Short, medium, and long taxi rides

We have already looked at taxi rides and have even (in exercise 30) looked at short, medium, and long taxi rides. However, in that exercise, we considered the distance traveled. In this exercise, we look at taxi rides from the perspective of how much time the ride took. Specifically, I want you to

1. Load taxi data from July 2019 into a data frame, using only the columns `tpep_ pickup_datetime`, `tpep_dropoff_datetime`, `passenger_count`, `trip_distance`, and `total_amount`, making sure to load `tpep_pickup_datetime` and `tpep_dropoff_datetime` as `datetime` columns.
2. Create a new column, `trip_time`, containing the amount of time each taxi ride took as a `timedelta`.
3. Determine the number and percentage of rides that took less than 1 minute.
4. Determine the average fare paid by people taking these short trips.
5. Determine the number and percentage of rides that took more than 10 hours.
6. Create a new column, `trip_time_group`, in which the values are `short` (< 10 minutes), `medium` (≥ between 10 minutes and 1 hour), and `long` (> 1 hour).
7. Determine the proportion of rides in each group.

8. For each value in `trip_time_group`, determine the average number of passengers.

Working it out

This exercise starts similarly to many others involving the taxi data. But whereas we were previously willing to let pandas determine the `dtype` of each column on its own, here we need to tell it to parse two of the columns as `Timestamp` objects. We could, of course, have imported them as text (i.e., the default) and then run `pd.to_timestamp` on them, but the following approach makes the process easier and cleaner. We can say

```
filename = '../data/nyc_taxi_2019-07.csv'

df = (
    pd
    .read_csv(filename,
              usecols=['tpep_pickup_datetime',
                       'tpep_dropoff_datetime',
                       'trip_distance',
                       'passenger_count',
                       'total_amount'],
              parse_dates=['tpep_pickup_datetime',
                           'tpep_dropoff_datetime'])
)
```

Notice that we need to include the two timestamp columns, `tpep_pickup_datetime` and `tpep_dropoff_datetime`, both in the `usecols` list and in the `parse_dates` list. In addition, this only works without any additional hints or tuning because the taxi dates are all stored in an unambiguous format of `YYYY-MM-DD`.

We can double-check that the columns have been interpreted correctly by invoking the `dtypes` method on our data frame:

```
df.dtypes
```

The result makes it clear that the parsing succeeded:

```
tpep_pickup_datetime     datetime64[ns]
tpep_dropoff_datetime    datetime64[ns]
passenger_count                 float64
trip_distance                   float64
total_amount                    float64
dtype: object
```

If we had not parsed the two timestamp columns, they would be listed as `object`, which as we've seen indicates that pandas is leaving them as

Python objects—most often, as strings.

With these timestamp columns in place, we can create a new `timedelta` column called `trip_time` by subtracting the pickup time from the dropoff time:

```
df['trip_time'] = (
    df['tpep_dropoff_datetime'] -
    df['tpep_pickup_datetime']
)
```

With this `timedelta` column in place, we can now ask questions about our data. For example, how many of the taxi rides in July 2019 took less than 1 minute?

To answer this, we need to perform a comparison with our `trip_time` column. We could create a `timestamp` object with `pd.to_timestamp`, but it turns out that pandas takes pity on us and allows us to compare a `timestamp` column with a string by doing the conversion behind the scenes:

```
df['trip_time'] < '1 minute'
```

This returns a new boolean series indicating when the trip took less than 1 minute. We can (as always) apply the boolean series to `df.loc` as a mask index, getting only those short trips:

```
df.loc[
    df['trip_time'] < '1 minute',    ①
    'trip_time'                      ②
    ].count()                        ③
```

① Row selector, for trips that took less than 1 minute

② Column selector, asking for only trip_time

③ Returns the number of non-NaN rows

We find that 70,212 taxi rides were less than 1 minute long. This seems like a large number of taxi rides to be taking so little time, but New York is a big city. What percentage of rides does this represent? We can find out by dividing this into the total number of rides in our data set:

```
df.loc[
    df['trip_time'] < '1 minute',
```

```
        'trip_time'
        ].count() / df['trip_time'].count() * 100
```

Just over 1% of taxi rides take less than a minute. That seems high to me, but maybe people enjoy taking a taxi for one or two blocks when they're in New York.

How much, on average, did people pay for those super-short taxi rides? To calculate that, we apply the mask index to `total_amount` and then calculate the mean (figure 10.2):

```
df.loc[
    df['trip_time'] < '1 minute',
    'total_amount'
    ].mean()
```

The result? More than $30! The only thing odder about so many people taking 1-minute taxi rides is the fact that they had to pay more than $30 for the privilege.

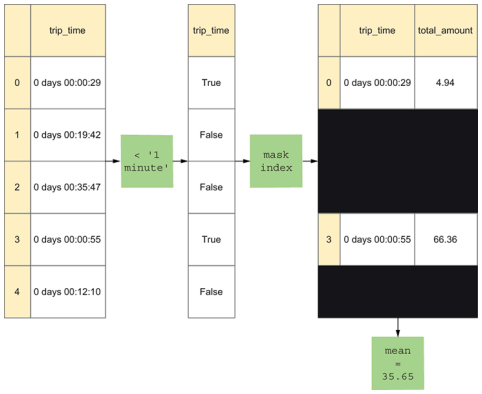


Figure 10.2 Illustration of how we got the mean of rides less than one minute long

Next, I asked you to find taxi rides that took more than 10 hours. I cannot imagine spending 10 hours in the back of a New York City taxi (or any other taxi, for that matter), but I thought it might be interesting to determine just how many such rides exist in this data set. Once again, we compare the `trip_time` column to a string:

```
df['trip_time'] > '10 hours'
```

We apply the resulting boolean series as a mask index and get all the long rides, which we then count:

```
df.loc[df['trip_time'] > '10 hours', 'trip_time].count()
```

Our data set contains 16,698 rides that took more than 10 hours. Seems high to me, but

maybe it's a reasonable percentage. Let's calculate that:

```
df.loc[df[
    'trip_time'] > '10 hours',
    'trip_time].count() / df['trip_time'].count() * 100
```

These rides constitute only 0.2% of all taxi rides. Even so, that means 2 out of every 1,000 taxi rides in New York take more than 10 hours.

Next, we want to group taxi rides into three categories: short, medium, and long. To do that, we can use `pd.cut`, a method we've already used to perform a similar task. But for this to work, we need to pass a `bins` value to `pd.cat` consisting of values that can be compared with our series.

Our intermediate cut points are 10 minutes and 1 hour: we call "short" trips those up to 10 minutes long, "medium" trips between 10 minutes and 1 hour, and "long" trips longer than 1 hour. However, `pd.cut` won't let us use strings to compare with our `timedelta` column. We thus need to create a Python list (or pandas series) of `timedelta` objects.

We do this with a list comprehension, a standard Python technique that is used more rarely by data analysts:

```
[pd.to_timedelta(arg)
 for arg in ['0 seconds', '10 minutes',
             '1 hour', '100 hours']]
```

In short, this list comprehension does the following:

1. Iterates over a list of strings
2. Converts each string to a `timedelta`
3. Returns a list of four `timedelta` objects based on the strings

We can then pass this list to `pd.cut`:

```
df['trip_time_group'] = (
    pd.cut(
           df['trip_time'],
           bins=[pd.to_timedelta(arg)
                 for arg in ['0 seconds',
                             '10 minutes',
                             '1 hour',
                             '100 hours']],
          labels=['short', 'medium', 'long'])
)
```

Notice that to have three labels, we need four cut points, or *bins* as they're

known here. And although we don't have to provide labels, we definitely should do so. The result of invoking `pd.cut` is a new series, which we then assign to `df['trip_time_group']`.

With those categories in place, we can perform a `groupby` query to see if there's any substantial difference in the number of passengers between short, medium, and long trips:

```
df.groupby('trip_time_group')['passenger_count'].mean()
```

Although short and medium trips both have average passenger counts of 1.5, there's a slightly larger average (1.7) for longer trips. That may imply that trips longer than 1 hour have more passengers, although it's hard to say why.

Solution

```
filename = '../data/nyc_taxi_2019-07.csv'

df = pd.read_csv(filename,
                 usecols=['tpep_pickup_datetime',
                          'tpep_dropoff_datetime',
                          'trip_distance', 'passenger_count',
                          'total_amount'],
                  parse_dates=['tpep_pickup_datetime',
```

```
                          'tpep_dropoff_datetime'])                    ①

df['trip_time'] = df['tpep_dropoff_datetime'
    ] - df['tpep_pickup_datetime']                                     ②

df.loc[df['trip_time'] < '1 minute', 'trip_time'].count()              ③
df.loc[df['trip_time'] < '1 minute', 'trip_time'
    ].count() / df['trip_time'].count()_ 100                           ④
df.loc[df['trip_time'] < '1 minute', 'total_amount'].mean()            ⑤

df.loc[df['trip_time'] > '10 hours', 'trip_time'].count()              ⑥
df.loc[df['trip_time'] > '10 hours', 'trip_time'
    ].count() / df['trip_time'].count()_ 100                           ⑦

df['trip_time_group'] = (
    pd.cut(
           df['trip_time'],
           bins=[pd.to_timedelta(arg)
                 for arg in ['0 seconds',
                             '10 minutes',
                             '1 hour',
                             '100 hours']],                            ⑧
          labels=['short', 'medium', 'long'])                          ⑨
)

df.groupby('trip_time_group')['passenger_count'].mean()                ⑩
```

① Loads the file, parsing two columns as timestamps

② Subtracts one timestamp from another, assigning to a new timedelta column

③ Counts trips less than 1 minute long

④ Calculates the percentage of trips less than 1 minute long

⑤ Applies the boolean index to the amount and takes the mean

⑥ Counts trips more than 10 hours long

⑦ Calculates the percentage of trips more than 10 hours long

⑧ Uses a list comprehension with to_timedelta to create bins

⑨ Assigns three labels for four edges in pd.cut

⑩ What was the mean number of passengers for short, medium, and long trips?

You can explore a version of this in the Pandas Tutor at **http://mng.bz/W1og**.

Beyond the exercise

- The data set you loaded is supposed to be for July 2019. How many trips are not from July 2019? That is, how many records are in the wrong file?
- What was the mean trip time for each number of passengers?

- Load taxi data from July 2019 and 2020. For each year, and then for each number of passengers, what was the mean amount paid?

xercise 40 • Writing dates, reading dates

In the previous exercise, we saw how easily we can read a CSV file into pandas, even if it includes date and time information. Generally speaking, we can tell the `parse_ dates` keyword argument which columns should be passed to `pd.to_datetime`, and we don't have to think about it any more. Sometimes, though, we're forced to deal with nonstandard date and time formats. We may be asked to write data using a particular format or (even more commonly) to read data that doesn't conform to a standard that pandas recognizes.

Fortunately, we can customize the ways in which datetime information is written to disk as well as how it is parsed when we read it into pandas.

In this exercise, we'll practice doing exactly that:

1. Load taxi data from July 2019 into a data frame, using only the columns `tpep_pickup_datetime`, `passenger_count`, `trip_distance`, and `total_amount`, making sure to load `tpep_pickup_datetime` as `datetime`.

2. Export this data frame to a tab-delimited CSV file. However, the datetime information should be written in the format `day/month/year HHh:MMm:SSs`. That is,
    1. The day should be a two-digit number.
    2. The month should be a two-digit number.
    3. The year should be a four-digit number.
    4. The hours should be a two-digit number, using a 24-hour clock, followed by the letter `h`.
    5. The minutes should be a two-digit number followed by the letter `m`.
    6. The seconds should be a two-digit number followed by the letter `s`.
    7. Read the CSV file you just created into a data frame. Be sure to parse the datetime column appropriately.

Using this weird format, the datetime February 3, 2023 at 10:11:12 is formatted as

```
03/02/23 10h:11m:12s
```

Working it out

This exercise is meant to give you some practice exporting and importing CSV files using alternative date formats. Most of the times I've had to read (or write) CSV files, dates have been in standard formats that pandas could parse without trouble. But there are always oddball logfiles that need parsing, typically written by custom programs, that use nonstandard formats. The good news is, we can use a custom format when working with CSV files.

I've had occasion to use this functionality in pandas just to translate files from one datetime format to another. In other words, I used pandas not for data analysis but instead as a very fancy date-translation service. That may feel like using a sledgehammer to swat a fly, but it got the job done and required me to write almost no code.

We started the exercise by importing New York taxi data from July 2019, including the `tpep_pickup_datetime` column. To ensure that

`tpep_pickup_datetime` is treated as a datetime column, we specify `parse_dates` to `read_csv`:

```
filename = '../data/nyc_taxi_2019-07.csv'
df = pd.read_csv(
    filename,
    usecols=['tpep_pickup_datetime',
             'trip_distance',
             'passenger_count',
             'total_amount'],
     parse_dates=['tpep_pickup_datetime'])
```

With the data frame in place, we can export the data to a CSV file containing only these four columns, but with the oddball datetime format that we used above. In theory, we could create a new column based on `tpep_pickup_datetime` but with the format we want and then export that new column to the CSV file. But it turns out pandas is one step ahead of us here, allowing us to specify the format in which datetime columns are written by passing a value to the `date_format` parameter.

The format is specified using `%` signs, using the format specifiers from `time.strftime` and described at

http://mng.bz/84DK. Our output can contain any combination of hours, minutes, months, days, time zones, and other elements. The format I described for the output is unusual in that dates are specified as `%d/%m/%Y`, meaning two digits for the day, two digits for the month, and four digits for the year, followed by a space character, and then the time in 24-hour format, but with `h` after the hours, `m` after the minutes, and `s` after the seconds. We can specify that as follows:

```
'%d/%m/%Y %Hh:%Mm:%Ss'
```

We can then write to our CSV file as follows:

```
df.to_csv('ex40_taxi_07_2019.csv',
          sep='\t',
          columns=['tpep_pickup_datetime', 'passenger_count',
                   'trip_distance', 'total_amount'],
          date_format='%d/%m/%Y %Hh:%Mm:%Ss')
```

In this code, we write to the file named ex40_taxi_07_2019.csv and specify (with the `sep` keyword argument) that we will use tabs to separate the fields. Pandas uses the `date_format`

parameter to indicate how all datetime columns (only `tpep_pickup_ datetime`, in our case) should be written.

Given that we're going to use this special datetime format in a number of places in our program, it's wiser to define it as a global string variable, `dt_format`. Then we can access that variable both within our call to `df.to_csv` and also later, in our date-parsing function. In such a case, the code looks like this:

```
dt_format='%d/%m/%Y %Hh:%Mm:%Ss'

df.to_csv('ex40_taxi_07_2019.csv',
          sep='\t',
          columns=['tpep_pickup_datetime',
                   'passenger_count',
                   'trip_distance',
                   'total_amount'],
          date_format=dt_format)
```

Once the file is written, I asked you to import it back into pandas, into a new data frame, and then to check that the reloaded `tpep_pickup_datetime` column remains a `datetime` column despite its weird date format.

NOTE Previous to pandas 2.0, you were encouraged to handle odd date formats in `read_csv` by passing a function to the `date_parser` keyword argument. That has been deprecated in favor of passing a string to `date_format`.

We can do that by invoking `df.read_csv`, specifying the filename, separator, columns, which column requires parsing as a date, and the date format (`dt_format`) we defined earlier:

```
df = pd.read_csv('ex40_taxi_07_2019.csv',
            sep='\t',
            usecols=['tpep_pickup_datetime',
                     'passenger_count',
                     'trip_distance',
                     'total_amount'],
            parse_dates=['tpep_pickup_datetime'],
            date_format=dt_format)
```

Solution

```
filename = '../data/nyc_taxi_2019-07.csv'
df = pd.read_csv(filename,                                    ①
                 usecols=['tpep_pickup_datetime',
                     'trip_distance',
                     'passenger_count', 'total_amount'],
                  parse_dates=['tpep_pickup_datetime'])

dt_format='%d/%m/%Y %Hh:%Mm:%Ss'                              ②
```

```
df.to_csv('ex40_taxi_07_2019.csv',                          ③
          sep='\t',
          columns=['tpep_pickup_datetime',
                   'passenger_count',
                   'trip_distance',
                   'total_amount'],
          date_format=dt_format)

import time

def parse_weird_format(s):                                  ④
    return pd.to_datetime(s, format=dt_format)

df = (
    pd
    .read_csv('ex40_taxi_07_2019.csv',                      ⑤
              sep='\t',
              usecols=['tpep_pickup_datetime',
                       'passenger_count',
                       'trip_distance',
                       'total_amount'],
            parse_dates=['tpep_pickup_datetime'],
            date_format=dt_format)                          ⑥
)
```

① Reads in the CSV file,
including the datetime column

② Specifies the format for
reading and writing

③ Writes the CSV file

④ This function takes a string
argument and returns a
datetime object.

⑤ Reads the CSV file, parsing
dates with our function

⑥ date_format expects a string it can use for parsing

You can explore a version of this in the Pandas Tutor at **http://mng.bz/E9Mq**.

Beyond the exercise

- Export the `tpep_pickup_datetime` date in Unix time—i.e., the number of seconds since 1 January 1970. This is an integer value.
- Read the data frame from this question back into a data frame. Read the `tpep_pickup_datetime` column into a string column, and use `pd.to_datetime` to convert it into a datetime column.
- Compare the speed of parsing time in `read_csv` versus doing so in a separate `to_datetime` step.

Time series

We have seen that a data frame's index can be an integer or string. But things get really exciting when we set a timestamp column to be

our index. In the pandas world, we call that a *time series*. When we create a time series, we can take advantage of a number of useful pandas features.

First, let's create a time series. We create a data frame with the dates and designations of NASA's Apollo program missions, grabbed from Wikipedia:

```
all_dfs = pd.read_html('https://en.wikipedia.org/wiki/Apollo_program')
df = all_dfs[2].copy()[['Date', 'Designation']]
```

Here is what the Wikipedia page looks like for me, with the table we want to retrieve:

Contents [hide]
(Top)
Background
NASA expansion
Choosing a mission mode
Spacecraft
Launch vehicles
Astronauts
Lunar mission profile
Development history
Mission summary
Samples returned
Costs
Apollo Applications Program
Recent observations
Legacy
Depictions on film
See also
References
Further reading
External links

Main article: Canceled Apollo missions

Several missions were planned for but were canceled before details were finalized.

Mission summary [edit]

For a more comprehensive list, see List of Apollo missions.

Designation	Date	Launch vehicle	CSM	LM	Crew
AS-201	Feb 26, 1966	AS-201	CSM-009	None	None
AS-203	Jul 5, 1966	AS-203	None	None	None
AS-202	Aug 25, 1966	AS-202	CSM-011	None	None
AS-204 (Apollo 1)	Feb 21, 1967	AS-204	CSM-012	None	Gus Grissom Ed White Roger B. Chaffee
Apollo 4	Nov 9, 1967	AS-501	CSM-017	LTA-10R	None
Apollo 5	Jan 22–23, 1968	AS-204	None	LM-1	None
Apollo 6	Apr 4, 1968	AS-502	CM-020 SM-014	LTA-2R	None
Apollo 7	Oct 11–22, 1968	AS-205	CSM-101	None	Wally Schirra Walt Cunningham Donn Eisele
Apollo 8	Dec 21–27, 1968	AS-503	CSM-103	LTA-B	Frank Borman James Lovell William Anders
Apollo 9	Mar 3–13, 1969	AS-504	CSM-104 *Gumdrop*	LM-3 *Spider*	James McDivitt David Scott Russell Schweickart
Apollo 10	May 18–26, 1969	AS-505	CSM-106 *Charlie Brown*	LM-4 *Snoopy*	Thomas Stafford John Young Eugene Cernan
Apollo 11	Jul 16–24, 1969	AS-506	CSM-107 *Columbia*	LM-5 *Eagle*	Neil Armstrong Michael Collins Buzz Aldrin
Apollo 12	Nov 14–24, 1969	AS-507	CSM-108 *Yankee Clipper*	LM-6 *Intrepid*	C. "Pete" Conrad Richard Gordon Alan Bean
Apollo 13	Apr 11–17, 1970	AS-508	CSM-109	LM-7	James Lovell Jack Swigert

Wikipedia page about the Apollo program

Some of the dates describe a single day (e.g., “Jul 5, 1966”) but others have ending dates, as well (e.g., “Jan 22–23, 1968”). We remove the ending dates where they appear in the text, to create a series of Apollo launch dates:

```
df['Date'] = pd.to_datetime(df['Date'].str.replace(
        '(-.+)?,', '', regex=True))
```

We then set the `Date` column to be the data frame’s index:

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

From this point on, `df` is a time series. We can see this by looking at `df.index`:

```
DatetimeIndex(['1966-02-26', '1966-07-05',
               '1966-08-25', '1967-02-21',
               '1967-11-09', '1968-01-22',
               '1968-04-04', '1968-10-11',
               '1968-12-21', '1969-03-03',
               '1969-05-18', '1969-07-16',
               '1969-11-14', '1970-04-11',
               '1971-01-31', '1971-07-26',
               '1972-04-16', '1972-12-07'],
              dtype='datetime64[ns]', name='Date', freq=None)
```

The index contains a number of `datetime` objects. With this in place, we can retrieve a row on a particular date,

just as we would with a normal index:

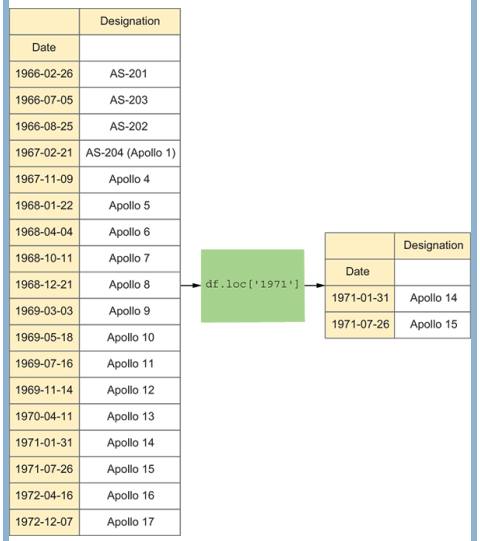
```
df.loc['1970-04-11']
```

Better yet, we can specify the smaller (i.e., more specific) parts of a date. That is, we can leave out the day, thus retrieving all values in a single month:

```
df.loc['1970-07']
```

Or we can specify just a year, thus getting all missions in that year:

```
df.loc['1971']
```



Retrieving rows of a time series via the year

We can also retrieve a set of rows with a slice by specifying starting and ending dates:

```
df.loc['1968-07-01':'1972-08-31']
```

Perhaps the most interesting and powerful feature of a time series is *resampling*. Resampling is similar to a `groupby` query, except that instead of producing one result per value of a categorical column, we get one result per chunk of time, starting at the earliest point in time and ending with the latest one. For example, resampling allows us to retrieve the mean value for every day, or every 2 weeks, or every 6 months, or every year of a data frame. For example, we can determine how many missions there were in every 6-month period covered by our data set:

```
df.resample('6M').count()
```

If the data is numeric, we can also run other aggregation

methods, such as `mean` and `std`.

xercise 41 • Oil prices

In this exercise, we work with a CSV file containing oil prices—specifically, West Texas Intermediate oil prices. These prices have been reported and updated daily, at least in our data set, from January 2, 1986 through the present day. (I constructed the CSV file using a Python program downloaded from **https://github.com/datasets/oil-prices**, which retrieves publicly available oil-price information from the US government.) In this exercise, we'll look at historical oil prices, using the datetime functionality in pandas to make such queries easier. Specifically, I want you to

1. Import the wti-daily.csv file into a data frame in which the `Date` column is treated as a datetime value and is set to be the index.

2. Answer these questions:
    1. What was the average price of a barrel of oil in June 1992?
    2. What was the average price of a barrel of oil in all of 1987?
    3. What was the average price from September 2003 through July 2014?
3. Show the price of oil at the end of each quarter in the data set.
4. For each year in the data set, show the average price.
5. On which date were oil prices the highest? When were they the lowest?

Working it out

We've already seen that by using the `dt` accessor, we can retrieve various parts of a datetime column. With that tool at our disposal, we can query our data in all sorts of ways. But we've also seen that certain queries can be easier to read and write when we change the index. This is particularly true when we set the index to be a datetime value. In this exercise, we explore a number of these functions while looking at historical oil prices.

The first task, as always, is to load the data from a CSV file into a data frame. In this case, the CSV file contains only two columns named `Date` and `Price`. In loading the CSV file into memory, we ask pandas to treat the `Date` column as (not surprisingly) a datetime value. We also ask it to make that column the index, via the `index_col` parameter:

```
filename = '../data/wti-daily.csv'

df = pd.read_csv(filename,
                 parse_dates=['Date'],
                 index_col=['Date'])
```

With that in place, we can make some queries. For starters, we want to determine the average price of oil during the month of June 1992. As usual, we can retrieve items from the data frame by using the `loc` accessor along with the index value that's of interest to us. But because it's a time series, we can provide a subset of the date, removing more specific parts to match a larger number of rows. We can thus say

```
df.loc['1992-06-15']
```

and get back the row for June 15, 1992. (If there were more than one row, we would get all of them back. But we know that each date in this data set is unique.) However, we're interested in all days in June 1992. We can thus say

```
df.loc['1992-06']
```

By leaving off the day, pandas matches and retrieves all rows in which the date is sometime in June 1992. To get the mean price during that period, we can say

```
df.loc['1992-06'].mean()
```

We get a result of just over $22.38.

I similarly asked you to find the mean price during 1987. Just as we can leave off the day to find all rows from a particular year and month, we can leave off the day and month to get all rows from a particular year:

```
df.loc['1987'].mean()
```

This retrieves all rows with a year of 1987. We then run `mean` on the `Price` column, which returns a number just over $19.20.

Next, I asked you to find the average price from September 2003 through July 2014. The easiest way to do this, when we have a time series, is to use a slice. Normally, Python slices are specified as a starting value and then 1 past the final value. For example, if we have a sequence (string, list, or tuple) named `s` and request `s[10:20]`, the slice retrieves values from the index 10 up to (but not including) 20.

Slices with a time series are similar, but as with other non-numeric indexes in pandas, we *include* the end of the slice. We can specify the date on which we want to start and also the date on which we want to end:

```
df.loc['2003-09':'2014-07']
```

This retrieves all rows from `df` starting September 1, 2003, and going through July 31,

2014. We can then run `mean` on the values we get back:

```
df.loc['2003-09':'2014-07'].mean()
```

This returns a value of just over $76.45.

Next, I asked you to find the price of oil at the end of each quarter in the data set. Pandas makes this easy to do with the `is_quarter_end` attribute on the `dt` accessor for `datetime` series. In our case, the `datetime` values aren't exactly in a series; they're on our index. How can we invoke `is_quarter_end` on our datetime index?

It turns out that we can invoke it directly on the index, getting a boolean series back:

```
df.index.is_quarter_end
```

This boolean series can then be applied as a mask index to `df`:

```
df.loc[df.index.is_quarter_end]
```

The result is a one-column data frame whose index

values are the final days of each quarter, regardless of whether it's the 30th or 31st of the month in question.

I also asked you to find the mean price of oil for each year in our data set. This is most easily accomplished by using `resample`, which is a kind of `groupby` but for time series: it lets us run an aggregation method (e.g., `mean`) for all the values in a given time period. If the time period doesn't exist in the data frame or is cut off, it still appears in the output to ensure that we have all the periods from start to finish.

When we run `resample`, we tell it what time-period granularity we want, giving a number and a letter representing the measurement. For example, we can run our aggregation method on a weekly basis with `1W` or on a bimonthly basis with `2M`. In this exercise, I asked to see annual average prices, which means specifying `1Y`. The resulting query is

```
df.resample('1Y').mean()
```

The result from a `resample` query is always a data frame in which the index contains the values from the end of each period. The index in our result thus starts at `1986-12-31` and goes through `2021-12-31`. Note that even if we only have partial values for a year, we get the average amount for that year.

Finally, I asked you to determine on which dates we had the historically highest and lowest oil prices. There are numerous ways to accomplish this, but I think it's easiest to sort the values in the `Price` column and then retrieve the first and last values from the resulting sorted series. Remember that we can retrieve more than one value by passing a list of indexes to `loc` or `iloc` and that if we use `iloc` (which retrieves by position), we can ask for index 0 (the first item) and –1 (the final item):

```
df['Price'].sort_values().iloc[[0, -1]]
```

You may be surprised that the lowest price of oil in this data set is –$36.98, meaning you

could get paid to accept a barrel of oil. If this sounds odd, you're right; it was the result of a dramatic drop in oil demand at the start of the Covid-19 pandemic. There wasn't enough storage space for the oil that had already been extracted from the ground, resulting in this bizarre situation. Look it up—it's just one of the many economic oddities of the pandemic.

Note that if we want the dates on which the minimum and maximum values were found but not the prices themselves, we can use

```
df['Price'].agg(['idxmin', 'idxmax'])
```

The `idxmin` and `idxmax` return the index corresponding to the minimum and maximum values, respectively. The `agg` method lets us invoke more than one aggregation method. So this query asks to see the indexes for the lowest and highest values in `Price`. We get the indexes (i.e., dates) but not the values themselves.

Solution

```
filename = '../data/wti-daily.csv'

df = pd.read_csv(filename,
                 parse_dates=['Date'],
                 index_col=['Date'])

df.loc['1992-06'].mean()
df.loc['1987'].mean()
df.loc['2003-09':'2014-07'].mean()
df.loc[df.index.is_quarter_end]
df.resample('1Y').mean()
df['Price'].sort_values().iloc[[0, -1]]
```

You can explore a version of this in the Pandas Tutor at **http://mng.bz/NVlE**.

Beyond the exercise

- Use `resample` to find, for each quarter, the mean and standard deviations in price.
- In which quarter did you see the biggest increase in mean price from the previous quarter?
- What was the biggest percentage increase in oil prices across quarters?

xercise 42 • Best tippers

We've looked at New York taxi data a number of times, and now we'll use our time-related

knowledge to study them again. This time, we'll try to understand when people tip their taxi drivers more generously. (If you're not from the United States, you may not be familiar with the custom of tipping, often 15% or 20%, in addition to whatever a taxi meter says you officially need to pay. In many other countries, this practice is unexpected, rare, or even illegal.) In particular, I'd like you to

1. Import the taxi info from both January and July 2019. Include the following columns: `tpep_pickup_datetime`, `passenger_count`, `trip_distance`, `fare_amount`, `extra`, `mta_tax`, `tip_amount`, `tolls_amount`, `improvement_surcharge`, `total_ amount`, and `congestion_surcharge`.

2. Create a new column, `pre_tip_amount`, with all the payment columns except `total_amount` and `tip_amount`. (Note that `total_amount` is the sum of all the other payment columns, including `tip_amount`. It should be equivalent to calculating `total_amount` - `tip_amount`.)
3. Create a new column, `tip_percentage`, showing the percentage of `pre_tip_amount` that the tip was.
4. Answer these questions:
    1. What was the mean tip percentage across all trips in the data set?
    2. How many times did people tip more than the pretip amount?
    3. On which day of the week do people tip the greatest percentage of the fare, on average?
    4. At which hour do people tip the greatest percentage?
    5. Do people typically tip more in January or July?
    6. What was the 1-day period in our data set when people tipped the greatest percentage?

Working it out

In this exercise, we ask the same question—when do people tip taxi drivers the most?—in a number of different ways. All of them use the extensive support for dates and times that pandas offers.

For starters, we load the data from both January and July 2019. As we've done before, we use a list comprehension along with `pd.read_csv`. This creates a list of data frames that we can turn into a single data frame with `pd.concat`:

```
filenames = ['../data/nyc_taxi_2019-01.csv',
             '../data/nyc_taxi_2019-07.csv']

all_dfs = [pd.read_csv(one_filename,
           usecols=['tpep_pickup_datetime',
                    'passenger_count',
                    'trip_distance',
                    'fare_amount','extra','mta_tax',
                    'tip_amount','tolls_amount',
                    'improvement_surcharge',
                    'total_amount','congestion_surcharge'],
           parse_dates=['tpep_pickup_datetime'])
           for one_filename in filenames]

df = pd.concat(all_dfs)
```

I asked you to include a large number of columns when creating the data frame so we

can calculate the tip percentage more accurately. I considered not asking you to specify `usecols` but rather to read all the data anyway—but as tempting as it may be to do that, it's not a good habit to get into. You should specify the columns you want in your data frame; otherwise, you'll find yourself running out of memory when you work with large data sets.

With our data frame in place, we want to calculate the pretip amount—that is, the amount on which the tip is based—for each ride. It's not always obvious what should (and shouldn't) be included in the tip. For example, do we include tolls for bridges and tunnels in our calculation? How about the surcharge that's sometimes added because the streets of New York are extra congested during those hours? For our purposes, we included all these fees and charges.

I thus asked you to create a new column, `pre_tip_amount`, that is the sum of six columns. How can we do that?

One possibility is to explicitly name those columns and add them together:

```
df['pre_tip_amount'] = (df['fare_amount'] +
                        df['extra'] +
                        df['mta_tax'] +
                        df['tolls_amount'] +
                        df['improvement_surcharge'] +
                        df['congestion_surcharge'])
```

This will certainly work, but it seems wordy. Perhaps there's a way to name the columns and sum them. The `sum` method would seem to be a perfect way to do this, except that it sums the rows rather than the columns. But wait! Many pandas methods allow us to specify the axis on which they run—and sure enough, `sum` is one of them. We can thus sum our selected columns by specifying `axis='columns'`:

```
df['pre_tip_amount'] = df[['fare_amount',
                           'extra',
                           'mta_tax',
                           'tolls_amount',
                           'improvement_surcharge',
                           'congestion_surcharge']
          ].sum(axis='columns')
```

Notice that we select our six columns with a list of strings.

We then run `sum` on these columns, producing a new pandas series. We assign this series back to `df['pre_tip_amount']`.

With that in hand, we're ready to create another column, `tip_percentage`, which contains the percentage of the pretip charge the user added as a tip:

```
df['tip_percentage'] = df['tip_amount'] / df['pre_tip_amount']
```

Our data frame now has all the information we need to answer our questions about tipping in New York taxis. For starters, what was the mean tipping rate across all taxi rides in our data set? We can find that by running the `mean` method on our `tip_percentage` column:

```
df['tip_percentage'].mean()
```

The answer is 13%. That seems low to me, so perhaps we're calculating the pretip base amount differently than others do. But maybe the data set is more complex than a straight percentage. For

example, does anyone tip more than 100%? We can find out:

```
(df['tip_percentage'] > 1).value_counts()
```

Here, we use `value_counts` to find how many people tipped more than 100% of the pretip amount. By applying `value_counts` to a boolean series, we're able to determine how often the `True` value is returned, meaning how often our condition is met.

The number of people giving above-and-beyond tips isn't overwhelming. But it's not zero, either, which came as a surprise to me. However, this number will skew the average tip upward. Perhaps there are people who aren't tipping at all, which will skew things downward. Let's take a look, calculating the percentage of riders who don't tip:

```
(df['tip_percentage'] == 0).value_counts(normalize=True)
```

Again, we use `value_counts` —but this time, we pass it `normalize=True` to get a percentage answer. And the

results are surprising, at least to me: about 32% of taxi riders in New York don't tip at all! This almost certainly has an effect on the mean tipping rate.

Next, we were curious to know whether people tip more on any particular day of the week. To do this, we combine `groupby` with the `dt` accessor's `day_of_week` attribute, which returns the integer for the day of the week, with Monday being 0 and Sunday being 6. You may think we need to define a new `day_of_week` column in our data frame so we can run a `groupby` on it. But no, the pandas developers make it possible to run a `groupby` not only on a column but also on the result we get back from `dt.day_of_week`:

```
df.groupby(df['tpep_pickup_datetime'].dt.day_of_week)
```

For each day of the week, we want to get the mean tip percentage. We thus run the following query:

```
df.groupby(df['tpep_pickup_datetime'].dt.day_of_week
    )['tip_percentage'].mean()
```

This gives us values, one for each day of the week. Just to make sure we get the right data, we then sort the resulting values from highest to lowest:

```
df.groupby(df['tpep_pickup_datetime'].dt.day_of_week
    )['tip_percentage'].mean().sort_values(ascending=False)
```

Much to my surprise, the tipping percentages aren't that different from one another. I was sure, before analyzing this data, that people tip more on weekends, but the data doesn't support that. On the contrary, it shows that people tip the least on Fridays and Saturdays and the most on Tuesdays and Wednesdays. However, the difference isn't that great, so I'm not sure if we can draw significant conclusions. Certainly if I were a taxi driver deciding which shifts to take, the tip amount on a given day wouldn't make much difference. (And besides, one-third of passengers aren't going to tip anything, right?)

But maybe the hour of the day makes a difference: that is, perhaps people tip better in the mornings or afternoons. I

thus asked you to create such a query, to find out at which hour of the day people tip the most on average:

```
df.groupby(df['tpep_pickup_datetime'].dt.hour
    )['tip_percentage'].mean().sort_values(ascending=False)
```

The query in this case is similar to the previous one. Here, however, we get more interesting results: people tip about 11% early in the morning (between 3:00 and 6:00 a.m.) and nearly 14% at night (from 8:00 to 11:00 p.m.). We see similar, if slightly lower, rates from 7:00 to 9:00 a.m.—so if you're unsure whether to take the 5:00 a.m. or 9:00 a.m. slot as a taxi driver, I'd suggest, based on average tips alone, choosing the latter.

Let's ask another question, which our data set can help us answer: do people tip more during the winter or the summer? (Or is there no difference?) We have data from both January and July, which should provide useful insights. We can say

```
df.groupby(df['tpep_pickup_datetime'].dt.month
    )['tip_percentage'].mean().sort_values(ascending=False)
```

The highest tips (20% on average) are given in May, followed by August, March, and September, respectively.

But wait a moment: our data set is supposed to contain data from January and July. How did other months get in there? The answer, of course, is that no data set is completely clean. Whether the dates are wrong, were reported late, or were otherwise scrambled along the way, our data contains information from other months. If we only compare January with July from this data set, we see a slight difference between the months, with tipping in January at 13.7% but in July at 12.1%. Whether that's because of summer tourists (who may —I'm just guessing—tip less) or people feeling more open with their cash during the winter months, I'm not sure.

Next, I asked what one-day period in the data set had the highest average percentage of tipping. This type of problem

is most easily solved with a time series, meaning we use a datetime value as the index:

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

With that in place, we can use `resample` with an argument of `1D` (i.e., one day) to find the day on which people tipped the most. First, we find the mean tipping percentage for each day in the time series:

```
df.resample('1D')['tip_percentage'].mean()
```

That works, but we'd like to sort these values so we can find the highest-tipping day. We do this by running `sort_values` on our results and then listing only the top 10 dates:

```
df.resample('1D')['tip_percentage'
    ].mean().sort_values(ascending=False).head(10)
```

The results include zero days from either January or July. Let's try this again but first get rid of dates that aren't in January or July:

```
df = pd.concat([df['2019-01-01':'2019-01-31'],
           df['2019-07-01':'2019-07-31']])
```

```
df.resample('1D')['tip_percentage'].mean().sort_values(
        ascending=False).head(10)
```

Having cleaned the data from non-January/July rows, we can see that all 10 of the highest-tipping days are in January. In our data sample, at least, people are more likely to tip better in the winter than in the summer. We can double-check by resampling at one-month granularity:

```
df.resample('1M')['tip_percentage'].mean().dropna()
```

Because we have only two months of data, but they're in January and July, using `resample` means we get `NaN` values for February, March, April, May, and June. We thus remove those with `dropna`. And we see that the average tipping rate in January is 13.7%, whereas in July it's 12.1%—a finding I hadn't anticipated.

Solution

```
filenames = ['../data/nyc_taxi_2019-01.csv',
             '../data/nyc_taxi_2019-07.csv']

all_dfs = [pd.read_csv(one_filename,                    ①
           usecols=['tpep_pickup_datetime',
                    'passenger_count',
```

```
                    'trip_distance',
                    'fare_amount','extra',
                    'mta_tax','tip_amount',
                    'tolls_amount',
                    'improvement_surcharge',
                    'total_amount','congestion_surcharge'],
            parse_dates=['tpep_pickup_datetime'])
            for one_filename in filenames]                          ②

df = pd.concat(all_dfs)                                             ③

df['pre_tip_amount'] = df[['fare_amount', 'extra',
                           'mta_tax', 'tolls_amount',
                           'improvement_surcharge',
                           'congestion_surcharge']].sum(
                axis='columns')                                     ④

df['tip_percentage'] = df[
    'tip_amount'] / df['pre_tip_amount']                            ⑤

df['tip_percentage'].mean()                                         ⑥

(df['tip_percentage'] > 1).value_counts()                           ⑦

(df['tip_percentage'] == 0).value_counts(
    normalize=True)                                                 ⑧

df.groupby(df[
    'tpep_pickup_datetime'].dt.day_of_week)[
    'tip_percentage'].mean().sort_values(ascending=False)           ⑨

df.groupby(df[
    'tpep_pickup_datetime'].dt.hour)[
    'tip_percentage'].mean().sort_values(
        ascending=False).head(5)                                    ⑩

df.groupby(df[
    'tpep_pickup_datetime'].dt.month)[
        'tip_percentage'].mean().sort_values(
            ascending=False)                                        ⑪

df = df.set_index('tpep_pickup_datetime')                           ⑫
```

```
  df.resample('1D')[
      'tip_percentage'].mean().sort_values(
          ascending=False).head(10)                                  ⑬

  df = pd.concat([df['2019-01-01':'2019-01-31'],
              df['2019-07-01':'2019-07-31']])                        ⑭

  df.resample('1D')[
      'tip_percentage'].mean().sort_values(
          ascending=False).head(10)
```

① Runs read_csv on each filename

② Our list comprehension returns a list of data frames

③ Combines the list of data frames into a single one

④ Creates the column pre_tip_amount

⑤ Calculates the tip percentage

⑥ What was the mean tip percentage across all trips?

⑦ How many trips tipped more than 100%?

⑧ What percentage of taxi riders give no tip at all?

⑨ Grouping by the day of week, calculates the mean tip percentage and then sorts

⑩ Grouping by the hour of the day, calculates the mean tip percentage and then sorts

⑪ Grouping by month, finds the mean tip percentage

⑫ Sets the data frame's index, making it a time series

⑬ Finds, for each day, the mean tip percentage

⑭ Excludes dates that aren't from January or July 2019

You can explore a version of this in the Pandas Tutor at **http://mng.bz/lW42**.

Beyond the exercise

- You saw that 32% of riders don't tip at all. Of those who *do*, what percentage do they tip, on average?
- How many of the rides in the data set, supposedly from January and July 2019, are from outside of those dates?
- Looking only at dates in January and July, in what week did passengers tip the greatest percentage?

ummary

In this chapter, we explored various ways pandas lets us examine data that includes a date-and-time component. We saw how to read datetime information into a data frame, extract datetime information from an existing column, break such a column apart, and interpret odd datetime formats. We also learned to create and work with a time series—a data frame in which a datetime column serves as our index—and how to query it in various ways, including resampling, letting us run aggregation methods over particular time periods.