Date and time data types

September 22, 2022

1 Dates and times in Python

In addition to common data types like strings, integers and booleans, Python also has date and time data types. You'll probably come across these most often via the built-in `datetime` module, so that's what we'll start with in this notebook.

Here's what we'll cover in this notebook: - Creating new dates and times - Parsing dates and times from text - Formatting dates as text - Calculating the difference between two datetimes - Working with dates in pandas

Let's start by importing the `datetime` object from the `datetime` module – this will allow us to create dates *and* times tied to a particular date, which is a common use.

[ ]:
```
from datetime import datetime
```

1.0.1 Creating new dates and times

The `datetime` object expects arguments in this order: year, month, day, hour, minute, second, microsecond, and `tzinfo`, a keyword argument for passing in timezone information. Let's create a date object for Feb. 8, 2021.

[ ]:
```
our_date = datetime(2021, 2, 8)
```

[ ]:
```
print(our_date)
```

You can access attributes of this date now, like `year` and `month`:

[ ]:
```
our_date.year
```

[ ]:
```
our_date.month
```

We could also make a specific time on that day – say, 1:30 p.m.:

[ ]:
```
our_datetime = datetime(2021, 2, 8, 13, 30)
```

[ ]:
```
print(our_datetime)
```

We could also make it 1:30 p.m. *EST*:

(Dealing with timezones in Python can be a huge pain, even for uncomplicated data, so it's usually easier to use a third-party library like `maya` or `pytz` – which we'll use here – instead of manually calculating timezone offsets with a `datetime.timezone` object.)

[ ]:
```
# conventionally, you'd group all imports at the top of the script
import pytz
```

[ ]:
```
est = pytz.timezone('America/New_York')
```

[ ]:
```
# you can get a list of all available timezones by running this cell
pytz.all_timezones
```

[ ]:
```
our_datetime_with_tz = datetime(2021, 2, 8, 13, 30, tzinfo=est)
```

[ ]:
```
print(our_datetime_with_tz)
```

If you need to get the datetime of *now* – i.e., when the script is run – you can use the handy method `now()`.

[ ]:
```
datetime.now()
```

1.0.2 Parsing dates and times from text

Let's say we've got a list of dates stored as strings (a common thing), and we want to make them into dates for future analysis:

[ ]:
```
our_dates = [
    '2018-09-10',
    '2018-10-30',
    '2017-03-13',
    '2000-01-02'
]
```

We can use datetime's `strptime` method for this. It expects two arguments: - The string to parse into a date - The *pattern* of the dates

In this case, the pattern is: four-digit year, dash, two-digit month, dash, two-digit day. According to the little mini-language of directives used to represent each piece of a date, that translates to `'%Y-%m-%d'`.

I can never remember these, so I have strftime.org bookmarked as a reference, and I check it *all the time*.

[ ]:
```
for d in our_dates:
    native_date = datetime.strptime(d, '%Y-%m-%d')
    print(native_date, type(native_date))
```

1.0.3 Formatting dates as text

You can use the `strftime` method to turn date objects into strings, formatted however you like and using the same mini-language of date directives we used with `strptime`. For these examples, we'll use the `our_date` variable.

[ ]:
```
our_date
```

[ ]:
```
our_date.strftime('%Y-%m-%d')
```

[ ]:
```
our_date.strftime('%m/%d/%Y')
```

[ ]:
```
our_date.strftime('It is %A, %B %-d, Year of our Lord %Y')
```

1.0.4 Calculating the difference between two datetimes

How old was each victim when they died? For each criminal case, how much time elapsed between the indictment and the sentencing? On average, how soon after the market-moving tweet did investors begin dumping stock in the company?

"Date diff" questions like this show up all the time in newsrooms. If you're using Python to work with your data, you can use basic math to yield a `datetime.timedelta` object and calculate the difference between two datetimes.

(For more complex data, using a third-party library like `dateutil` would be worth your while.)

But let's start with a simple example: How many minutes elapsed between two dates?

[ ]:
```
datetime1 = datetime(2021, 2, 8, 13, 30)
datetime2 = datetime(2021, 2, 10, 16, 30)
```

[ ]:
```
datetime2 - datetime1
```

We can access the `seconds` attribute of the `timedelta` object that gets returned from this math problem:

[ ]:
```
elapsed_time = datetime2 - datetime1
print(elapsed_time.seconds)
```

… and if we want minutes, just divide by 60.

[ ]:
```
print(elapsed_time.seconds / 60)
```

`timedelta` objects are also useful for "what was the date 160 days ago"-type problems:

[ ]:
```
from datetime import timedelta

rn = datetime.now()
date_248_days_ago = rn - timedelta(days=248)

print(date_248_days_ago)
```

[ ]:
```
# two weeks ago
rn - timedelta(weeks=2)
```

[ ]:
```
# 89382 seconds ago
rn - timedelta(seconds=89382)
```

Date math can get a little complicated (time zones! leap years! birthdays!), but if you get stuck it's probable that someone on the Internet has solved your problem already.

1.0.5 Working with dates in pandas

Let's take a look at a couple of things you might want to do when working with dates in pandas: Parsing values as dates when you *import* the data into a dataframe, and coercing *existing* data to dates.

Let's import pandas and load up some congressional junkets data (`../data/congress_junkets.csv`) with date values in two of the columns (`DepartureDate` and `ReturnDate`).

We're going to specify the `parse_dates` argument when we call the `read_csv()` function – we'll hand this keyword argument a *list* of columns to parse as dates.

[ ]:
```
import pandas as pd
```

[ ]:
```
df = pd.read_csv('../data/congress_junkets.csv',
                 parse_dates=['DepartureDate', 'ReturnDate'])
```

[ ]:
```
df.head()
```

This method can be slow if you have a lot of data; one way to speed it up is to supply pandas with the correct date format using the specification language mentioned above (here's an example from StackOverflow).

We'll use a lambda expression here for brevity, but you can also define a new function and pass that in as the `date_parser` argument, as well.

[ ]:
```
df = pd.read_csv('../data/congress_junkets.csv',
                 parse_dates=['DepartureDate', 'ReturnDate'],
                 date_parser=lambda x: datetime.strptime(x, '%m/%d/%Y'))
```

Much quicker. We can verify that the column is now a date column by checking the data frame's `dtypes` attribute:

[ ]:
```
df.dtypes
```

Sometimes, for Reasons™, it makes more sense to do the conversion after you've loaded your data. Let's use the same data but do the conversion *after* it's in the dataframe.

[ ]:
```
df2 = pd.read_csv('../data/congress_junkets.csv')
```

[ ]:
```
df2.head()
```

To do this, we'll create a new column for each date with the `to_datetime()` method. We'll hand it three things: - The column to convert to a datetime - The expected `format` of the dates (m/d/y, in this case) - What to do if the parser runs into an error – instead of throwing an error, we'll `coerce` (errors will become `NaN`)

[ ]:
```
df2['departure_date_new'] = pd.to_datetime(df2['DepartureDate'],
                                           format='%m/%d/%Y',
                                           errors='coerce')

df2['return_date_new'] = pd.to_datetime(df2['ReturnDate'],
                                        format='%m/%d/%Y',
                                        errors='coerce')
```

[ ]:
```
df.head()
```