Manipulate Times, Dates, and Time Spans

Table of Contents

Introduction

Dealing with dates and times in Python can be a hassle. Thankfully, there's a built-in way of making it easier: The Python datetime module.

`datetime` helps us identify and process time-related elements like dates, hours, minutes, seconds, days of the week, months, years, etc. It offers various services like managing time zones and daylight savings time. It can work with timestamp data. It can extract the day of the week, day of the month, and other date and time formats from strings.

In short, it's a really powerful way of handling anything date and time related in Python. So, let's get into it!

Python datetime Classes

Before jumping into writing code, it's worth looking at the five main object classes that are used in the `datetime` module. Depending on what we're trying to do, we'll likely need to make use of one or more of these distinct classes:

- **datetime** – Allows us to manipulate times and dates together (month, day, year, hour, second, microsecond).
- **date** – Allows us to manipulate dates independent of time (month, day, year).
- **time** – Allows us to manipulate time independent of date (hour, minute, second, microsecond).
- **timedelta**— A *duration* of time used for manipulating dates and measuring.
- **tzinfo**— An abstract class for dealing with time zones.

If those distinctions don't make sense yet, don't worry! Let's dive into `datetime` and start working with it to better understand how these are applied.

Creating Date Objects

First, let's take a closer look at a `datetime` object. Since `datetime` is both a module and a class within that module, we'll start by importing the `datetime` class from the `datetime` module.

Then, we'll print the current date and time to take a closer look at what's contained in a `datetime` object. We can do this using `datetime`'s `.now()` function. We'll print our datetime object, and then also print its type using `type()` so we can take a closer look.

```
# import datetime class from datetime module
from datetime import datetime

# get current date
datetime_object = datetime.now()
print(datetime_object)
print('Type :- ',type(datetime_object))
```

Output:

```
2019-10-25 10:24:01.521881
Type :-
```

We can see from the results above that `datetime_object` is indeed a `datetime` object of the `datetime` class. This includes the year, month, day, hour, minute, second, and microsecond.

Extract Year and Month from the Date

Now we've seen what makes up a `datetime` object, we can probably guess how `date` and `time` objects look, because we know that `date` objects are just like `datetime` without the time data, and `time` objects are just like `datetime` without the date data.

We can also antipate some problems. For example, in most data sets, date and time information is stored in string format! Also, we may not *want* all of this date and time data — if we're doing something like a monthly sales analysis, breaking things down by microsecond isn't going to be very useful.

So now, let's start digging into a common task in data science: extracting only the elements that we actually want from a string using `datetime`.

To do this, we need to do a few things.

Handling Date and Time Strings with strptime() and strftime()

Thankfully, `datetime` includes two methods, `strptime()` and `strftime()`, for converting objects from strings to `datetime` objects and vice versa. `strptime()` can read strings with date and time information and convert them to `datetime` objects, and `strftime()` converts datetime objects back into strings.

Of course, `strptime()` isn't magic — it can't turn *any* string into a date and time, and it will need a little help from us to interpret what it's seeing! But it's capable of reading most conventional string formats for date and time data (see the documentation for more details). Let's give it a date string in YYYY-MM-DD format and see what it can do!

```
my_string = '2019-10-31'
my_date = datetime.strptime(my_string, "%Y-%m-%d")  # yyyy-mm-dd

print(my_date)
print('Type: ',type(my_date))
```

Output:

```
2019-10-31 00:00:00
Type:
```

Note that `strptime()` took two arguments: the string (`my_string`) and `"%Y-%m-%d"`, another string that tells `strptime()` how to interpret the input string `my_string`. `%Y`, for example, tells it to expect the first four characters of the string to be the year.

A full list of these patterns is available in the documentation, and we'll go into these methods in more depth later in this tutorial.

You may also have noticed that a time of `00:00:00` has been added to the date. That's because we created a `datetime` object, which must include a date *and* a time. `00:00:00` is the default time that will be assigned if no time is designated in the string we're inputting.

Anyway, we were hoping to separate out specific elements of the date for our analysis. One way can do that using the built-in class attributes of a datetime object, like `.month` or `.year`:

```
print('Month: ', my_date.month) # To Get month from date
print('Year: ', my_date.year) # To Get month from year
```

Output:

```
Month:  10 Year:  2019
```

Getting Day of the Month and Day of the Week from a Date

Let's do some more extraction, because that's a really common task. This time, we'll try to get the day of the month and the day of the week from `my_date`. Datetime will give us the day of the week as a number using its `.weekday()` function, but we can convert this to a text format (i.e. Monday, Tuesday, Wednesday…) using the `calendar` module and a method called `day_name`.

We'll start by importing `calendar`, and then using `.day` and `.weekday()` on `my_date`. From there, we can get the day of the week in text format like so:

```
# import calendar module
import calendar
print('Day of Month:', my_date.day)

# to get name of day(in number) from date
print('Day of Week (number): ', my_date.weekday())

# to get name of day from date
print('Day of Week (name): ', calendar.day_name[my_date.weekday()])
Day of Month: 31 Day of Week (number):  3 Day of Week (name):  Thursday
```

Wait a minute, that looks a bit odd! The third day of the week should be Wednesday, not Thursday, right?

Let's take a closer look at that `day_name` variable using a for loop:

```
j = 0
for i in calendar.day_name:
    print(j,'-',i)
    j+=1
```

Output:

```
0 - Monday 1 - Tuesday 2 - Wednesday 3 - Thursday 4 - Friday 5 - Saturday 6 -
Sunday
```

Now we can see that Python starts weeks on Monday and counts from the index 0 rather than starting at 1. So, it makes sense that the number 3 is converted to "Thursday" as we saw above.

Getting Hours and Minutes From a Python Datetime Object

Now let's dig into time and extract the hours and minutes from datetime object. Much like what we did above with month and year, we can use class attributes `.hour` and `.minute` to get the hours and minutes of the day.

Let's set a new date and time using the `.now()` function. As of this writing, it's October 25, 2019 at 10:25 AM. You'll get different results depending on when you choose to run this code, of course!

```
from datetime import datetime todays_date = datetime.now()

# to get hour from datetime
print('Hour: ', todays_date.hour)

# to get minute from datetime
print('Minute: ', todays_date.minute)
```

Output:

```
Hour:  10 Minute:  25
```

Getting Week of the Year from a Datetime Object

We can also do fancier things with `datetime`. For example, what if we want to know what week of the year it is?

We can get the year, week of the year, and day of the week from a `datetime` object with the `.isocalendar()` function.

Specifically, `isocalendar()` returns a tuple with ISO year, week number and weekday. The ISO calendar is a widely-used standard calendar based on the Gregorian calendar. You can read about it in more detail at that link, but for our purposes, all we need to know is that it works as a regular calendar, starting each week on Monday.

```
# Return a 3-tuple, (ISO year, ISO week number, ISO weekday).
todays_date.isocalendar()
```

Output:

```
(2019, 43, 5)
```

Note that in the ISO calendar, the week starts counting from 1, so here 5 represents the correct day of the week: Friday.

We can see from the above that this is the 43rd week of the year, but if we wanted to isolate that number, we could do so with indexing just as we might for any other Python list or tuple:

```
todays_date.isocalendar()[1]
```

Output:

```
43
```

Measuring Time Span with Timedelta Objects

Often, we may want to measure a span of time, or a duration, using Python datetime. We can do this with its built-in `timedelta` class. A `timedelta` object represents the amount of time between two dates or times. We can use this to measure time spans, or manipulate dates or times by adding and subtracting from them, etc.

By default a timedelta object has all parameters set to zero. Let's create a new timedelta object that's two weeks long and see how that looks:

```
#import datetime
from datetime import timedelta
# create timedelta object with difference of 2 weeks
d = timedelta(weeks=2)

print(d)
print(type(d))
print(d.days)
```

Output:

```
14 days, 0:00:00 <class 'datetime.timedelta'> 14
```

Note that we can get our time duration in days by using the `timedelta` class attribute `.days`. As we can see in its documentation, we can also get this time duration in seconds or microseconds.

Let's create another timedelta duration to get a bit more practice:

```
year = timedelta(days=365)
print(year)
```

Output:

```
365 days, 0:00:00
```

Now let's start doing using timedelta objects together with datetime objects to do some math! Specifically, let's add a few diffeent time durations to the current time and date to see what date it will be after 15 days, what date it was two weeks ago.

To do this, we can use the `+` or `-` operators to add or subtract the timedelta object to/from a datetime object. The result will be the datetime object plus or minus the duration of time specified in our timedelta object. Cool, right?

(Note: in the code below, it's October 25 at 11:12 AM; your results will differ depending on when you run the code since we're getting our `datetime` object using the `.now()` function).

```
#import datetime
from datetime import datetime, timedelta
# get current time
now = datetime.now()
print ("Today's date: ", str(now))

#add 15 days to current date
future_date_after_15days = now + timedelta(days = 15)
print('Date after 15 days: ', future_date_after_15days)

#subtract 2 weeks from current date
two_weeks_ago = now - timedelta(weeks = 2)
print('Date two weeks ago: ', two_weeks_ago)
print('two_weeks_ago object type: ', type(two_weeks_ago))
```

Output:

```
Today's date:  2019-10-25 11:12:24.863308 Date after 15 days:  2019-11-09 11:12:24.863308 Date two weeks ago:  2019-10-11 11:12:24.863308 two_weeks_ago object type:  <class 'datetime.datetime'>
```

Note that the output of these mathematical operations is still a `datetime` object.

Find the Difference Between Two Dates and Times

Similar to what we did above, we can also subtract one date from another date to find the timespan between them using datetime.

Because the result of this math is a *duration*, the object produced when we subtract one date from another will be a `timedelta` object.

Here, we'll create two `date` objects (remeber, these work the same as `datetime` objects, they just don't include time data) and subtract one from the other to find the duration:

```
# import datetime
from datetime import date
# Create two dates
date1 = date(2008, 8, 18)
date2 = date(2008, 8, 10)

# Difference between two dates
delta = date2 - date1
print("Difference: ", delta.days)
print('delta object type: ', type(delta))
```

Output:

```
Difference:  -8 delta object type:  <class 'datetime.timedelta'>
```

Above, we used only dates for the sake of clarity, but we can do the same thing with `datetime` objects to get a more precise measurement that includes hours, minutes, and seconds as well:

```
# import datetime
from datetime import datetime
# create two dates with year, month, day, hour, minute, and second
date1 = datetime(2017, 6, 21, 18, 25, 30)
date2 = datetime(2017, 5, 16, 8, 21, 10)

# Difference between two dates
diff = date1-date2
print("Difference: ", diff)
```

Output:

```
Difference:  36 days, 10:04:20
```

Formatting Dates: More on strftime() and strptime()

We touched briefly on `strftime()` and `strptime()` earlier, but let's take a closer look at these methods, as they're often important for data analysis work in Python.

`strptime()` is the method we used before, and you'll recall that it can turn a date and time that's formatted as a text string into a datetime object, in the following format:
`time.strptime(string, format)`
Note that it takes two arguments:

- string – the time in string format that we want to convert
- format – the specific formatting of the time in the string, so that strptime() can parse it correctly

Let's try converting a different kind of date string this time. This site is a really useful reference for finding the formatting codes needed to help `strptime()` interpret our string input.

```
# import datetime
from datetime import datetime
date_string = "1 August, 2019"

# format date
date_object = datetime.strptime(date_string, "%d %B, %Y")

print("date_object: ", date_object)
```

Output:

```
date_object:  2019-08-01 00:00:00
```

Now let's do something a bit more advanced to practice everything we've learned so far! We'll start with a date in string format, convert it to a datetime object, and look at a couple different ways of formatting it (dd/mm and mm/dd).

Then, sticking with the mm/dd formatting, we'll convert it into a Unix timestamp. Then we'll convert it back into a `datetime` object, and convert *that* back into strings using a few different strftime patterns to control the output:

```
# import datetime
from datetime import datetime
dt_string = "12/11/2018 09:15:32"
# Considering date is in dd/mm/yyyy format
dt_object1 = datetime.strptime(dt_string, "%d/%m/%Y %H:%M:%S")
print("dt_object1:", dt_object1)
# Considering date is in mm/dd/yyyy format
dt_object2 = datetime.strptime(dt_string, "%m/%d/%Y %H:%M:%S")
print("dt_object2:", dt_object2)

# Convert dt_object2 to Unix Timestamp
timestamp = datetime.timestamp(dt_object2)
print('Unix Timestamp: ', timestamp)

# Convert back into datetime
date_time = datetime.fromtimestamp(timestamp)
d = date_time.strftime("%c")
print("Output 1:", d)
d = date_time.strftime("%x")
print("Output 2:", d)
d = date_time.strftime("%X")
print("Output 3:", d)
```

Output:

```
dt_object1: 2018-11-12 09:15:32 dt_object2: 2018-12-11 09:15:32 Unix Timestamp:  1544537732.0
Output 1: Tue Dec 11 09:15:32 2018 Output 2: 12/11/18 Output 3: 09:15:32
```

Here's an image you can save with a cheat sheet for common, useful strptime and strftime patterns:

Directive	Meaning
%a	Weekday as locale's abbreviated name.
%A	Weekday as locale's full name.
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.
%d	Day of the month as a zero-padded decimal number.
%b	Month as locale's abbreviated name.
%B	Month as locale's full name.
%m	Month as a zero-padded decimal number.
%y	Year without century as a zero-padded decimal number.
%Y	Year with century as a decimal number.
%H	Hour (24-hour clock) as a zero-padded decimal number.
%I	Hour (12-hour clock) as a zero-padded decimal number.
%p	Locale's equivalent of either AM or PM.
%M	Minute as a zero-padded decimal number.
%S	Second as a zero-padded decimal number.
%f	Microsecond as a decimal number, zero-padded on the left.
%z	UTC offset in the form ±HHMM[SS[.ffffff]] (empty string if the object is naive).

Let's get a little more practice using these:

```
# current date and time
now = datetime.now()

# get year from date
year = now.strftime("%Y")
print("Year:", year)

# get month from date
month = now.strftime("%m")
print("Month;", month)

# get day from date
day = now.strftime("%d")
print("Day:", day)

# format time in HH:MM:SS
time = now.strftime("%H:%M:%S")
print("Time:", time)

# format date
date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
print("Date and Time:",date_time)
```

Output:

```
Year: 2019 Month; 10 Day: 25 Time: 11:56:41 Date and Time: 10/25/2019, 11:56:41
```

Handling Timezones

Working with dates and times in Pythin can get even more complicated when timezones get involved. Thankfully, the `pytz` module exists to help us deal with cross-timezone conversions. It also handles the daylight savings time in locations that use that.

We can use the `localize` function to add a time zone location to a Python datetime object. Then we can use the function `astimezone()` to convert the existing local time zone into any other time zone we specify (it takes the time zone we want to convert into as an argument).

For example:

```
# import timezone from pytz module
from pytz import timezone
# Create timezone US/Eastern
east = timezone('US/Eastern')
# Localize date
loc_dt = east.localize(datetime(2011, 11, 2, 7, 27, 0))
print(loc_dt)

# Convert localized date into Asia/Kolkata timezone
kolkata = timezone("Asia/Kolkata")
print(loc_dt.astimezone(kolkata))

# Convert localized date into Australia/Sydney timezone
au_tz = timezone('Australia/Sydney')
print(loc_dt.astimezone(au_tz))
```

Output:

```
2011-11-02 07:27:00-04:00 2011-11-02 16:57:00+05:30 2011-11-02 22:27:00+11:00
```

This module can help make life simpler when working with data sets that include multiple different time zones.

Working with pandas Datetime Objects

Data scientists love `pandas` for many reasons. One of them is that it contains extensive capabilities and features for working with time series data. Much like `datetime` itself, pandas has both `datetime` and `timedelta` objects for specifying dates and times and durations, respectively.

We can convert date, time, and duration text strings into pandas Datetime objects using these functions:

- **to_datetime()**: Converts string dates and times into Python datetime objects.
- **to_timedelta()**: Finds differences in times in terms of days, hours, minutes, and seconds.

And as we'll see, these functions are actually quite good at converting strings to Python datetime objects by detecting their format automatically, without needing us to define it using strftime patterns.

Let's look at a quick example:

```
# import pandas module as pd
import pandas as pd
# create date object using to_datetime() function
date = pd.to_datetime("8th of sep, 2019")
print(date)
```

Output:

```
2019-09-08 00:00:00
```

Note that even though we gave it a string with some complicating factors like a "th" and "sep" rather than "Sep." or "September", pandas was able to correctly parse the string and return a formatted date.

We can also use pandas (and some of its affiliated numpy functionality) to create date ranges automatically as pandas Series. Below, for example, we create a series of twelve dates starting from the day we defined above. Then we create a different series of dates starting from a predefined date using `pd.date_range()`:

```
# Create date series using numpy and to_timedelta() function
date_series = date + pd.to_timedelta(np.arange(12), 'D')
print(date_series)

# Create date series using date_range() function
date_series = pd.date_range('08/10/2019', periods = 12, freq ='D')
print(date_series)
```

Output:

```
DatetimeIndex(['2019-09-08', '2019-09-09', '2019-09-10', '2019-09-11', '2019-09-12', '2019-09-13', '2019-09-14', '2019-09-15', '2019-09-16', '2019-09-17', '2019-09-18', '2019-09-19'], dtype='datetime64[ns]', freq=None) DatetimeIndex(['2019-08-10', '2019-08-11', '2019-08-12', '2019-08-13', '2019-08-14', '2019-08-15', '2019-08-16', '2019-08-17', '2019-08-18', '2019-08-19', '2019-08-20', '2019-08-21'], dtype='datetime64[ns]', freq='D')
```

Get Year, Month, Day, Hour, Minute in pandas

We can easily get year, month, day, hour, or minute from dates in a column of a pandas dataframe using `dt` attributes for all columns. For example, we can use `df['date'].dt.year` to extract only the year from a pandas column that includes the full date.

To explore this, let's make a quick DataFrame using one of the Series we created above:

```
# Create a DataFrame with one column date
df = pd.DataFrame()
df['date'] = date_series df.head()
```

Output:

	date
0	2019-08-10
1	2019-08-11
2	2019-08-12
3	2019-08-13
4	2019-08-14

Now, let's create separate columns for each element of the date by using the relevant Python datetime (accessed with `dt`) attributes:

```
# Extract year, month, day, hour, and minute. Assign all these date component to new column.
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute
df.head()
```

Output:

date	year	month	day	hour	minute	
0	2019-08-10	2019	8	10	0	0
1	2019-08-11	2019	8	11	0	0
2	2019-08-12	2019	8	12	0	0
3	2019-08-13	2019	8	13	0	0
4	2019-08-14	2019	8	14	0	0

Get Weekday and Day of Year

Pandas is also capable of getting other elements, like the day of the week and the day of the year, from its datetime objects. Again, we can use `dt` attributes to do this. Note that here, as in Python generally, the week starts on Monday at index 0, so day of the week 5 is *Saturday*.

```
# get Weekday and Day of Year. Assign all these date component to new column.
df['weekday'] = df['date'].dt.weekday
df['day_name'] = df['date'].dt.weekday_name
df['dayofyear'] = df['date'].dt.dayofyear
df.head()
```

Output:

	date	year	month	day	hour	minute	weekday	day_name	dayofyear
0	2019-08-10	2019	8	10	0	0	5	Saturday	222
1	2019-08-11	2019	8	11	0	0	6	Sunday	223
2	2019-08-12	2019	8	12	0	0	0	Monday	224
3	2019-08-13	2019	8	13	0	0	1	Tuesday	225
4	2019-08-14	2019	8	14	0	0	2	Wednesday	226

Convert Date Object into DataFrame Index

We can also use pandas to make a datetime column into the index of our DataFrame. This can be very helpful for tasks like exploratory data visualization, because matplotlib will recognize that the DataFrame index is a time series and plot the data accordingly.
To do this, all we have to do is redefine `df.index`:

```
# Assign date column to dataframe index
df.index = df.date
df.head()
```

Output:

date	year	month	day	hour	minute	weekday	day_name	dayofyear
date								
2019-08-10	2019-08-10	2019	8	10	0	0	5	Saturday	222
2019-08-11	2019-08-11	2019	8	11	0	0	6	Sunday	223
2019-08-12	2019-08-12	2019	8	12	0	0	0	Monday	224
2019-08-13	2019-08-13	2019	8	13	0	0	1	Tuesday	225
2019-08-14	2019-08-14	2019	8	14	0	0	2	Wednesday	226

Conclusion

In this tutorial, we've taken a deep dive into Python datetime, and also done some work with pandas and the calendar module. We've covered a lot, but remember: the best way to learn something is by actually writing code yourself!