Reference guide: Datetime manipulation

The following tables can serve as reference guides to remind you of the shorthand code for manipulating datetime strings into individual objects.

Manipulating datetime strings in Python

Below, you will find a table with the datetime functions you can use to help you manipulate datetime objects in different ways.

Code	Format	Example
%a	Abbreviated workday	Sun
%A	Weekday	Sunday
%b	Abbreviated month	Jan
%B	Month name	January
%c	Date and time	Sun Jan 1 00:00:00 2021
%d	Day (leading zeros)	01 to 31
%H	24 hours	00 to 23
%I	12 hours	01 to 12
%j	Day of year	001 to 366
%m	Month	01 to 12
%M	Minute	00 to 59
%p	AM or PM	AM/PM
%S	Seconds	00 to 61

%U	Week number (Sun)	00 to 53
%W	Week number (Mon)	00 to 53
%w	Weekday	0 to 6
%x	Locale's appropriate date representation	08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)
%X	A locale's appropriate time representation	21:30:00 (en_US); 21:30:00 (de_DE)
%y	Year without century	00 to 99
%Y	Year	2022
%z	Offset	+0900
%Z	Time zone	EDT/JST/WET etc (GMT)

Datetime functions to remember

All of the following date string manipulations require the datetime package to be imported first.

Code	Input Type	Input Example	Output Type	Output Example
`datetime.strptime("25/11/2022", "%d/%m/%Y")`	string	"25/11/2022"	DateTime	"2022-11-25 00:00:00"
`datetime.strftime(dt_object, "%d/%m/%Y")`	DateTime	"2022-11-25 00:00:00"	string	"25/11/2022"
`dt_object = datetime.strptime("25/11/2022", "%d/%m/%Y")` `datetime.timestamp(dt_object)`	string	"25/11/2022"	float (UTC timestamp in seconds)	1617836400.0
`datetime.strptime("25/11/2022", "%d/%m/%Y").strftime("%Y-%m-%d")`	string	"25/11/2022"	string	"2022-11-25"
`datetime.fromtimestamp(1617836400.0)`	float (UTC timestamp in seconds)	1617836400.0	DateTime	datetime.datetime(2021, 4, 7, 23, 0)
`datetime.fromtimestamp(1617836400.0).strftime("%d/%m/%Y")`	float (UTC timestamp in seconds)	1617836400.0	string	"'07/04/2021'"

`from pytz import timezone`
`ny_time = datetime.strptime("25-11-2022  09:34:00-0700", "%d-%m-%Y %H:%M:%S%z")`
`Tokyo_time = ny_time.astimezone(timezone('Asia/Tokyo'))`
	string	NewYork timezone "25-11-2022 09:34:00-0700"	DateTime	Tokyo timezone 2022, 11, 26, 1, 34, JST+9:00:00 STD>

`datetime.strptime("20:00", "%H:%M").strftime("%I:%M %p")`	string	"20:00"	string	"08:00 PM"
`datetime.strptime("08:00 PM", "%I:%M  %p").strftime("%H:%M")`	string	"08:00 PM"	string	"20:00"

Datetime in NumPy and pandas

A preface regarding terminology in the following section: `datetime` refers to the specific module of that name in the Python standard library or to the specific class within that module.

Datetime (or uncapitalized, datetime) refers to any date/time-related object from any library or language.

You've learned that the `datetime` module in Python's standard library contains a number of classes used to work with time data, including `date`, `time`, `datetime`, `timedelta`, `timezone`, and `tzinfo`. Remember, modules are similar to libraries, in that they are groups of related classes and functions, but they are generally subcomponents of libraries. Classes are data types that bundle data and functionality together.

NumPy and pandas have their own datetime classes that offer significant performance boosts when working with large datasets. Pandas datetime classes, like the rest of the pandas library, are built on NumPy. These classes have very similar (and in many cases identical) functionality to Python's native datetime classes, but they run more efficiently due to NumPy and pandas' vectorization capabilities. Therefore, although you *can* use `datetime` data in pandas, it's generally better to use NumPy or pandas datetime objects when working in pandas, if possible.

NumPy's datetime classes include, most notably, `datetime64` and `timedelta64`. Like `datetime` objects, `datetime64` objects contain date and time information in a single data structure; and, like `timedelta` objects, `timedelta64` objects contain information pertaining to spans of time.

Pandas' datetime classes include `Timestamp`, `Timedelta`, `Period`, and `DateOffset`. Because these classes are efficient and dynamic in their capabilities, you often don't need to import the `datetime` module when working with datetime data in pandas. Also, pandas will automatically recognize datetime-like data and convert it to the appropriate class when possible. Here's an example:

```
data = ['2023-01-20', '2023-04-27', '2023-06-15']
my_series = pd.Series(data)
my_series
0    2023-01-20
1    2023-04-27
2    2023-06-15
dtype: object
```

This series contains string data, but it can be converted to `datetime64` data using the `pd.to_datetime()` function:

```
my_series = pd.to_datetime(my_series)
my_series
```

```
0   2023-01-20
1   2023-04-27
2   2023-06-15
dtype: datetime64[ns]
```

Refer to the [pandas to_datetime() documentation] for more information about this function.

When a `Series` object contains datetime data, you can use `dt` to access various properties of the data. For example:

```
print(my_series.dt.year)
print()
print(my_series.dt.month)
print()
print(my_series.dt.day)
```

```
0    2023
1    2023
2    2023
dtype: int64

0    1
1    4
2    6
dtype: int64

0    20
1    27
2    15
dtype: int64
```

Note that it's not uncommon to import the `datetime` module from Python's standard library as `dt`. You may have encountered this yourself. In such case, `dt` is being used as an alias. The

pandas `dt` Series accessor (as demonstrated in the last example) is a different thing entirely. Refer to the pandas dt accessor documentation for more information.

Key takeaways

Use reference guides like the tables above throughout your career to help remind you of the different ways to manipulate datetime objects. Even experts in the field use reference guides, rather than memorizing all this information. Getting familiar with guides like these will be beneficial because you will be using them throughout your career as a data professional.