Time and Date Series in Python I

This tutorial will provide an overview of some of the functionality Python, NumPy and Pandas offer to manipulate time series, and date, time objects. The tutorial is meant to be an introduction and it will not be exhaustive in the treatment of the exstensive functionality offered by the python libraries.

Learning goals

- `Python` - Date and Time objects
- `NumPy` - Date and Time objects
- `Pandas` - Date and Time objects

Prerequisites

- Python and NumPy
- Pandas, DataFrames and Series

Python time and dates

Python offers a series of core and additional functions dealing with time, time zones, calendars, etc.

These modules come standard with Python3. The modules mostly provide access to the Linux/Unix (or Windows) OS time-oriented functionality.

Module Name	Description	Usage
Time	This module provides various time-related functions.	`import time`
datetime	This module supplies classes for manipulating dates and times.	`import datetime`
Calendar	This module allows you to output calendars like the Unix cal program, and provides additional useful functions related to the calendar.	`import calendar`
Zoneinfo	A time zone implementation interfacing with the system's time zone data (if available e.g. Unix/Linux; see also PEP 615).	`import zoneinfo`

Although we will cover some of the functionality in these python modules, an exhustive coverage of the functionality of these python modules is beyond the scope of the current tutorial. That said below the list of modules pertaining with time and date operations and data.

Python Core Time Objects

The core functionality of time events in Python can be accessed using `time` and `datetime` .

For example, we can import `datetime` and access my date of birth we could easily access it (if you were to know the correct one) use syntax as follows:

In [1]:
```
from datetime import datetime

my_birthdate = datetime(year=1985,
                        month=11,
                        day=15,
                        hour=11,
                        minute=36,
                        second=55)
print(my_birthdate)
```

```
1985-11-15 11:36:55
```

As you can see above, `datetime` allows addressing years, months, days, etc. just like properties of objects. The output variables are well formatted strings.

The months, days, hours and minutes are formatted in ways that we are generally used to, the 'look good'. The formatting used by `datetime` is called ISO 8601 format (it follows this syntax: `YYYY-MM-DDTHH:MM:SS. mmmmmm`).

`time` is also part of python's core functionality.

It provides a more fexible, yet more complex, less object-oriented and less data-science friendly interface.

For example, if we wanted to get a date and time (say the current time) using `time` , we would need to first import two modules, `gmtime` to get the time and `strftime` to format the time.

`gmtime` would allow extracting the information about the current time.

In [2]:
```
from time import gmtime, strftime
gmtime()
```

Out[2]:
```
time.struct_time(tm_year=2022, tm_mon=4, tm_mday=28, tm_hour=14, tm_min=42, tm_sec=15, t
m_wday=3, tm_yday=118, tm_isdst=0)
```

As you can appreciate, the output is not nicely formatted. At least not as nicely formatted using the ISO 8601 standard as seen above in the output of `datetime` .

We can format the output with some additional work, using `strftime()` . For example,

In [3]:
```
strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
```

Out[3]:
```
'Thu, 28 Apr 2022 14:45:34 +0000'
```

The type of the time value sequence returned by `gmtime()` It is an object with a named tuple interface with following values that can be accessed by index and by attribute name. The following values are present:

Index	Attribute	Values
0	tm_year	(for example, 1993)
1	tm_mon	range [1, 12]

Index	Attribute	Values
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]; see (2) in strftime() description
6	tm_wday	range [0, 6], Monday is 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 or -1; see below
N/A	tm_zone	abbreviation of timezone name
N/A	tm_gmtoff	offset east of UTC in seconds

Try, returning the values into a variable and access them as previously learned (hint, `vals.<val_name>`):

In [4]:
```
vals = gmtime()

vals.tm_hour
```

Out[4]:
```
14
```

Question:

Is the information extracted for you by `gmtime()` correct for your time zone? Why?

`gmtime()` returns time for `Greenwich Mean Time (GMT)` that is a standard timezone in the UK. Given that Austin, TX is currently in CDT. We could return the current hour in Austin by adding the 5 hours different.

For example:

In [6]:
```
vals = gmtime()

theHourInGreenwichUK = vals.tm_hour
print('Time in Greenwhich, UK: ', theHourInGreenwichUK)

theHourInAustinTX = theHourInGreenwichUK - 5
print('Time in Austin, TX: ', theHourInAustinTX)
```

```
Time in Greenwhich, UK:  14
Time in Austin, TX:  9
```

So, informastion can be extracted, yet, it takes some work and knowledge to get it out. It would be convenient to have easier ways to manipulate dates more explicitely.

A note about Naive and Aware objects.

Objects pertaining time and dates depend on geographical position and time zones. Time zones could be day-light savings, e.g., CDT for Austin, TX or standard, e.g., CST for Austin, TX.

Two types of objects exist in python:

- Aware Objects - Objects are called `aware` when they can locate themselves relative to other aware objects. Aware objects represent a specific moment in time that is not open to interpretation.

- Naive Objects - A naive objects do not contain enough information to unambiguously locate themselves relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass.

The outputs on `datetime` are `time` are naive by default. Optional calls to `timezone` can be made to make the objects aware. See here for more information about Naive and Aware time objects.

The functionality of `datetime` and `time` is exstensive as they allow advance and flexible manipulation of events, timing and calendars. The goal of this tutorial is to get you started and provide an overview of methods available to access time and date information.

One limitation of of the core function for time manipulation in python is the inability to handle large arrays of dates and times. This is an important feature, especially for data science applications where often times arrays can reach a large size.

This issue is similar to the issue we have encountered before, where the standard lists in Python are limited in their ability to support advanced numerical operations.

NumPy objects for date and time

As we learned in previous tutorials, NumPy addressed the needs of the community to handle large datasets by providing the ability to store large variables into computable objects.

Also for time and date functionality, NumPy added to the set of native date and time python functions, a set of additional functions to handle large objects, manipulate and perform numerical operations.

The functions NumPy added build on top of two NumPy objects:

- `datetime64` -
- `timedelta64`

`datetime64` and `timedelta64` encodes dates as 64-bit integers, array operator. This allows the data stored as a date to be much larger than the basic python date and time variables and allows them to be computable.

A NumPy `datetime64` variable can be usig very specific formats for example:

In [7]:
```
import numpy as np
date = np.array('1985-11-15', dtype=np.datetime64)
```

```
print('ISO format <', date, '>')
```

```
ISO format < 1985-11-15 >
```

Note that the NumPy arrays requires a very specific format as input: 'YYY-MM-DD'. This is the most basic way to create datetimes array using an ISO 8601 date or datetime format.

This new object is though fundamentally different than any of the previously encountered data, time objects in python. For example, we can perform direct computations using this object (datetime in NumPy works with most NumPy operations such as `arange`).

Hereafter we add numbers in the range between [0-11] to the days in the original date and replicate the entry in the array in a single operation:

In [8]:

```
date + np.arange(12)
```

Out[8]:

```
array(['1985-11-15', '1985-11-16', '1985-11-17', '1985-11-18',
       '1985-11-19', '1985-11-20', '1985-11-21', '1985-11-22',
       '1985-11-23', '1985-11-24', '1985-11-25', '1985-11-26'],
      dtype='datetime64[D]')
```

A range of days can be specified at definition to when defining the array:

In [10]:

```
daysInTwoMonths = np.arange('2020-02', '2020-04', dtype='datetime64[D]')

print('All days in two months', daysInTwoMonths)

# it's aware of the leap years because it uses my computer's calendar
```

```
All days in two months ['2020-02-01' '2020-02-02' '2020-02-03' '2020-02-04' '2020-02-05'
 '2020-02-06' '2020-02-07' '2020-02-08' '2020-02-09' '2020-02-10'
 '2020-02-11' '2020-02-12' '2020-02-13' '2020-02-14' '2020-02-15'
 '2020-02-16' '2020-02-17' '2020-02-18' '2020-02-19' '2020-02-20'
 '2020-02-21' '2020-02-22' '2020-02-23' '2020-02-24' '2020-02-25'
 '2020-02-26' '2020-02-27' '2020-02-28' '2020-02-29' '2020-03-01'
 '2020-03-02' '2020-03-03' '2020-03-04' '2020-03-05' '2020-03-06'
 '2020-03-07' '2020-03-08' '2020-03-09' '2020-03-10' '2020-03-11'
 '2020-03-12' '2020-03-13' '2020-03-14' '2020-03-15' '2020-03-16'
 '2020-03-17' '2020-03-18' '2020-03-19' '2020-03-20' '2020-03-21'
 '2020-03-22' '2020-03-23' '2020-03-24' '2020-03-25' '2020-03-26'
 '2020-03-27' '2020-03-28' '2020-03-29' '2020-03-30' '2020-03-31']
```

Question: Does NumPy treat "February' properly?

In the short overview of NumPy above we have seen that it made two fundamental improvements over the more basic ways to access date and time information in python:

- First it allowed handling larger arrays
- Second it made the date and time arrays computable, just like the other objects in NumPy

---

Pandas Time Series

As discussed in previous tutorials, Pandas was developed to handle data from investment banking. A key aspect of the data types that investment banking have to deal with is time series; data tht varies across actual time (seconds, minutes, hours, days, weeks, months and years).

Because of this, Pandas comes with abundant functionality to deal with time series, time events and computations. The Pandas time series functionality leverages what we have alredy discussed about the library *indexes*, *series* and *data frames* and builts on top of python's own time functions.

Whereas Python allows accessing fundamental time and date information from the operative system and NumPy turns the time and date objects into computable data, Pandas improves this functionality to build full-time series tables and objects.

Pandas consolidated functionality from the basic python libraries as well as NumPy (as well as scikits.timeseries). In doing so, Pandas has also created new functionality for manipulating time series data. Pandas Organizes and provides a cohesive interface to call NumPy and Python core time functions.

Pandas builds upon the following objects:

- Python's `datetime` (as well as `dateutil`)
- NumPy's `datetime64`
- Pandas'
  - `Timestamp`
  - `DatetimeIndex`
  - `Series`
  - `DataFrame`

Hereafer, we will dig into a few of the features pandas has to offer!

Pandas basic time an date objects

The fundamental objects dealing with time series in Pandas is `Timestamp`. The object can be created from heterogeneous input data formats, that are more user friendly.

This means that the same Timestamp can be created from differently formatted inputs. Pandas' will interpret the different formatting and map them to appropriate datetime objects (with some limits of course):

In [11]:
```
import pandas as pd
import numpy as np
```

In [12]:
```
date = pd.to_datetime("15th of November, 1985")
date
print('Output from "15th of November, 1985"', date)

date = pd.to_datetime("November 15, 1985")
print('Output from "November 15, 2015"     ', date)
```

```
Output from "15th of November, 1985" 1985-11-15 00:00:00
Output from "November 15, 2015"      1985-11-15 00:00:00
```

Another example of the flexibility of Pandas' interpretation of inputs:

In [13]:
```
date = pd.to_datetime([datetime(1985, 11, 15), '14th of November, 1985',
                       '1985-Nov-15', '07-11-1985', '19851108'])
date
```

Out[13]:
```
DatetimeIndex(['1985-11-15', '1985-11-14', '1985-11-15', '1985-07-11',
               '1985-11-08'],
              dtype='datetime64[ns]', freq=None)
```

All above diverse formats have been interpreted correctly. Instead, the following odd formatting will fail, because Pandas will not be prepared to parse it:

In [ ]:
```
date1 = pd.to_datetime("November 15 : 1985")
print('Output from "November 15 : 1985"     ', date1) # the colon is uninterpretable
```

`.to_datetime` is the method to take an input, interpret it and map it to the appropriate time and date (`Timestamp` in Pandas).

The `Timestamp` object is elevated in pandas into the `DatetimeIndex` object. The DatetimeIndex allows array-style operations, similar to those allowed by NumPay.

For example, we can show 20 days starting on November 15 1985:

In [17]:
```
date = pd.to_datetime("15th of November, 1985")
date + pd.to_timedelta(np.arange(20), 'D') # D means days, so we are adding 20 days
```

Out[17]:
```
DatetimeIndex(['1985-11-15', '1985-11-16', '1985-11-17', '1985-11-18',
               '1985-11-19', '1985-11-20', '1985-11-21', '1985-11-22',
               '1985-11-23', '1985-11-24', '1985-11-25', '1985-11-26',
               '1985-11-27', '1985-11-28', '1985-11-29', '1985-11-30',
               '1985-12-01', '1985-12-02', '1985-12-03', '1985-12-04'],
              dtype='datetime64[ns]', freq=None)
```

Note how the `Timestamp` object `date` was changed into a `DatetimeIndex` by the above operation.

An operation similar to the above can also be implemented by accessing directly Pandas' functionality to manipulate dates. This is another way to create a range starting from a secific year-month-day:

In [19]:
```
dates = pd.date_range("19851115", periods=20, freq='D')
dates
```

Out[19]:
```
DatetimeIndex(['1985-11-15', '1985-11-16', '1985-11-17', '1985-11-18',
               '1985-11-19', '1985-11-20', '1985-11-21', '1985-11-22',
               '1985-11-23', '1985-11-24', '1985-11-25', '1985-11-26',
               '1985-11-27', '1985-11-28', '1985-11-29', '1985-11-30',
```

```
               '1985-12-01', '1985-12-02', '1985-12-03', '1985-12-04'],
              dtype='datetime64[ns]', freq='D')
```

A simple change allows performing the same operation across months:

In [20]:

```
dates = pd.date_range("19851115", periods=20, freq='M')
dates
```

Out[20]:

```
DatetimeIndex(['1985-11-30', '1985-12-31', '1986-01-31', '1986-02-28',
               '1986-03-31', '1986-04-30', '1986-05-31', '1986-06-30',
               '1986-07-31', '1986-08-31', '1986-09-30', '1986-10-31',
               '1986-11-30', '1986-12-31', '1987-01-31', '1987-02-28',
               '1987-03-31', '1987-04-30', '1987-05-31', '1987-06-30'],
              dtype='datetime64[ns]', freq='M')
```

In [21]:

```
df = pd.DataFrame(np.random.randn(np.squeeze(dates.shape), 4),
                  index=dates,
                  columns={"Peace", "Love", "knowledge", "compassion"})
```

In [22]:

```
df
```

Out[22]:

	Love	Peace	knowledge	compassion
1985-11-30	-1.834546	-0.204266	1.531281	-0.402424
1985-12-31	1.011686	0.076477	-1.173525	-1.292698
1986-01-31	-2.458481	-0.291967	-0.711714	-0.469752
1986-02-28	-0.332149	0.811562	-0.039810	-1.505092
1986-03-31	1.357090	0.000153	0.787503	1.953151
1986-04-30	0.555078	-1.435789	-0.139448	-1.739449
1986-05-31	2.020613	0.420698	-0.039241	-0.605588
1986-06-30	0.457624	0.722869	-0.852043	1.047168
1986-07-31	-0.745445	0.727126	0.587263	1.309102
1986-08-31	-1.023602	-0.738582	0.246016	1.309080
1986-09-30	0.565017	0.637501	-0.728786	-1.162866
1986-10-31	0.873187	0.258498	-0.977004	-0.822623
1986-11-30	1.696537	-0.042741	-0.780107	-1.649245
1986-12-31	-0.520969	-1.438838	-1.247339	0.772518
1987-01-31	0.403368	0.132312	0.857444	-1.604900
1987-02-28	0.107720	-0.377668	1.315748	0.233390
1987-03-31	-2.027114	1.403452	0.911501	0.947739
1987-04-30	-0.979991	0.197548	-0.763724	2.784403
1987-05-31	0.301964	-1.655228	-0.729491	0.882616
1987-06-30	1.979494	-1.320755	0.941357	-1.070008

Manipulating Pandas' Time Series

We have learned before that Pandas Series and DataFrame objects allow sotring data and are indexed. Indices become extremely helpful when we start dealing with Time Series.

For example we can initialize a Pandas `index` object by using a Pandas `DatetimeIndex` object. The latter can be the used to label a series. Let's take a look.

We will first create a range of dates and use them as indeces to a Series:

In [23]:

```
index   = pd.date_range("19851115", periods=6, freq='Y')
data    = [0, 1, 2, 3, 4, 5]

dseries = pd.Series(data, index=index)
dseries
```

Out[23]:

```
1985-12-31    0
1986-12-31    1
1987-12-31    2
1988-12-31    3
1989-12-31    4
1990-12-31    5
Freq: A-DEC, dtype: int64
```

The above dseries is a Padas Series object with a frequency in years ('Y', aligned to december). The labels (or indices) are dates and the data are just numbers between 0 and 5.

Just like any label or index we can use the data indices to address the Pandas Series. For example, we can find the a subset of four entries:

In [24]:

```
dseries['1986-12-31':'1989-12-31']
```

Out[24]:

```
1986-12-31    1
1987-12-31    2
1988-12-31    3
1989-12-31    4
Freq: A-DEC, dtype: int64
```

Also Pandas allows indexing Series by leading trails of the labels. For example, the year can be used as index – without need to specify the rest of the date:

In [25]:

```
dseries['1985']
```

Out[25]:

```
1985-12-31    0
Freq: A-DEC, dtype: int64
```

This operation would have worked also if the dates had been different (not all matching December 31).

In sum, in this tutorial we have provided an overview of some of the functionality that Python, NumPy and Pandas offer to manipulate time series, and date, time objects. Tutorial with more advanced functionality for manipulating and platting time series.

Exercise:

Create a Pandas DataFrame (hah!) that uses dates as indices and stores data about three variables: 1. Mental health. 2. Quality of life. and 3. Heart rate. The measurements were taken in a single subject between May 19, 1985 and June 16, 1995 at a monthly rate.

In [32]:
```
# attemtped to create the index this way, but then I go over the time frame
index = pd.date_range("19850519", periods=132, freq='M')
index
```

Out[32]:
```
DatetimeIndex(['1985-05-31', '1985-06-30', '1985-07-31', '1985-08-31',
               '1985-09-30', '1985-10-31', '1985-11-30', '1985-12-31',
               '1986-01-31', '1986-02-28',
               ...
               '1995-07-31', '1995-08-31', '1995-09-30', '1995-10-31',
               '1995-11-30', '1995-12-31', '1996-01-31', '1996-02-29',
               '1996-03-31', '1996-04-30'],
              dtype='datetime64[ns]', length=132, freq='M')
```

In [35]:
```
# created an index in which the date changes by 28 days each time
index = np.arange('1985-05-19', '1995-06-16', 28, dtype='datetime64[D]')
index1 = pd.to_datetime(index)
index1
```

Out[35]:
```
DatetimeIndex(['1985-05-19', '1985-06-16', '1985-07-14', '1985-08-11',
               '1985-09-08', '1985-10-06', '1985-11-03', '1985-12-01',
               '1985-12-29', '1986-01-26',
               ...
               '1994-09-25', '1994-10-23', '1994-11-20', '1994-12-18',
               '1995-01-15', '1995-02-12', '1995-03-12', '1995-04-09',
               '1995-05-07', '1995-06-04'],
              dtype='datetime64[ns]', length=132, freq=None)
```

In [36]:
```
dict = {'Mental Health': 15 +3*np.random.randn(132),
       'Quality of Life': 15 + 3*np.random.randn(132),
       'Heart Rate' : 80 +15*np.random.randn(132)}
pd.DataFrame(dict, index=index)
```

Out[36]:

	Mental Health	Quality of Life	Heart Rate
1985-05-19	15.572341	13.976754	102.434679
1985-06-16	16.405139	13.151350	101.535502
1985-07-14	14.600681	15.930185	65.295665
1985-08-11	14.808037	16.629365	67.995322
1985-09-08	12.078402	10.611726	93.607990
...	...	...	...

	Mental Health	Quality of Life	Heart Rate
1995-02-12	13.057649	14.645331	71.680172
1995-03-12	13.018153	19.558863	44.721509
1995-04-09	19.241995	12.884520	88.560288
1995-05-07	19.250806	15.093485	86.817904
1995-06-04	9.972214	16.008059	84.026515

132 rows × 3 columns