Introduction

`pandas` can handle dates in several ways:

- comparison
- extraction of day, hour, etc...
- addition of delays
- etc...

Date objects in `pandas` are the one of `numpy`, which differ a bit from those of native Python but are nonetheless compatible with them.

Useful functions

Create a dates series

`pd.date_range` expects 3 out of 4 of the following parameters to be specified:

- `start` : (date) first date
- `end` : (date) last date
- `periods` (integer): number of values
- `freq` : elapsed time between 2 consecutive dates. Accepted values are given here.

```
In [1]: import pandas as pd
        # 7 values, once every 2 hours, from 21st of January 2023 at 30s past midnight
        pd.date_range(start='21/01/2023 00:00:30', freq='2h', periods=7)
```

```
Out[1]: DatetimeIndex(['2023-01-21 00:00:30', '2023-01-21 02:00:30',
                       '2023-01-21 04:00:30', '2023-01-21 06:00:30',
                       '2023-01-21 08:00:30', '2023-01-21 10:00:30',
                       '2023-01-21 12:00:30'],
                      dtype='datetime64[ns]', freq='2h')
```

```
In [2]: # a value every 3 days, from 21st of January 2023 to 4 of February
        pd.date_range(start='21/01/2023 00:00:00', end='04/02/2023 00:00:00', freq='3D')
```

```
Out[2]:   DatetimeIndex(['2023-01-21', '2023-01-24', '2023-01-27', '2023-01-30',
                         '2023-02-02', '2023-02-05', '2023-02-08', '2023-02-11',
                         '2023-02-14', '2023-02-17', '2023-02-20', '2023-02-23',
                         '2023-02-26', '2023-03-01', '2023-03-04', '2023-03-07',
                         '2023-03-10', '2023-03-13', '2023-03-16', '2023-03-19',
                         '2023-03-22', '2023-03-25', '2023-03-28', '2023-03-31'],
                        dtype='datetime64[ns]', freq='3D')
```

In [3]:
```python
# 3 values from 21st of January 2023 to 4 of February
from datetime import datetime, timedelta
start = datetime(day=21, month=1, year=2023)
end = datetime(day=4, month=2, year=2023)
pd.date_range(start=start, end=end, periods=3)
```

Out[3]:
```
DatetimeIndex(['2023-01-21', '2023-01-28', '2023-02-04'], dtype='datetime64
[ns]', freq=None)
```

With `pandas`, one can specify dates in 3 different ways:

- A string

  Very handy but beware of bad interpreation done by `pandas`. In particular, the English way of processing date is to write the month before the day (ex: 21th june --> 06/21).

- Python date-like objects: `datetime.datetime`, `datetime.timedelta`
- `numpy` date-like objects
- `pandas` date-like objects: `pd.TimeStamp`, `pd.TimeDelta`

Resample temporal data

Whenever temporal data comes with a too high or too low frequency, the `pd.resample` functions can modify this frequency to ease processing. There exists two cases:

- *upsampling*: additional values are added between current values, hence frequency is increased
- *downsampling*: existing values are aggragated following a specific rule, hence frequency is decreased

Upsampling

Here after, a DataFrame that has a datetime index: six values, one every 2 hours, from 15th of August.

```
In [4]: df = pd.DataFrame({'A': range(6), 'B': range(6, 12)},
                          index=pd.date_range(start='15/08/2024 00:00:00', freq='2h', periods
        df
```

Out[4]:

	A	B
2024-08-15 00:00:00	0	6
2024-08-15 02:00:00	1	7
2024-08-15 04:00:00	2	8
2024-08-15 06:00:00	3	9
2024-08-15 08:00:00	4	10
2024-08-15 10:00:00	5	11

Let's resample the DataFrame with one value every 50 min:

In [5]:
```python
rs = df.resample('50min')
```

`rs` is an instance of type `Resampler`. It must be used with a rule qui doit être appelé avec une règle that defines what to do with unexistant data. For instance `ffill`, for *'forward fill'*, fill missing values with the previous existing value.

In [6]:
```python
rs.ffill()
```

Out[6]:

	A	B
2024-08-15 00:00:00	0	6
2024-08-15 00:50:00	0	6
2024-08-15 01:40:00	0	6
2024-08-15 02:30:00	1	7
2024-08-15 03:20:00	1	7
2024-08-15 04:10:00	2	8
2024-08-15 05:00:00	2	8
2024-08-15 05:50:00	2	8
2024-08-15 06:40:00	3	9
2024-08-15 07:30:00	3	9
2024-08-15 08:20:00	4	10
2024-08-15 09:10:00	4	10
2024-08-15 10:00:00	5	11

Loading [MathJax]/extensions/Safe.js

Downsampling

Let's reduce the frequency from 2h to 4h. Values are averaged.

In [7]:
```
rs = df.resample('4h')
rs.mean()
```

Out[7]:

	A	B
2024-08-15 00:00:00	0.5	6.5
2024-08-15 04:00:00	2.5	8.5
2024-08-15 08:00:00	4.5	10.5

Loading [MathJax]/extensions/Safe.js

Indexing

With a temporal index, one can use slicing methods but with date-like instances:

In [8]:
```
start = datetime(day=15, month=8, year=2024, hour=3)
end = start + timedelta(hours=5)
print(start, end, sep='\n')
```

```
2024-08-15 03:00:00
2024-08-15 08:00:00
```

In [9]:
```
df.loc[start:end]   # everything from start to end
                    # both are included since `loc` is label-based
```

Out[9]:

	A	B
2024-08-15 04:00:00	2	8
2024-08-15 06:00:00	3	9
2024-08-15 08:00:00	4	10

`dt` accessor

When temporal data is stored in a column, the **`dt` accessor** provides date-specific methods:

In [10]:
```
df.index.name = 'date'
df = df.reset_index()
df
```

Out[10]:

	date	A	B
0	2024-08-15 00:00:00	0	6
1	2024-08-15 02:00:00	1	7
2	2024-08-15 04:00:00	2	8
3	2024-08-15 06:00:00	3	9
4	2024-08-15 08:00:00	4	10
5	2024-08-15 10:00:00	5	11

Loading [MathJax]/extensions/Safe.js

Let's have a look to all the methods and attributes of the `dt` object:

In [11]:
```python
# listing of attributes and methods of object dt
for attr in dir(df['date'].dt):
    if not attr.startswith('_'):
        print(attr, end=' / ')
```

```
as_unit / ceil / date / day / day_name / day_of_week / day_of_year / dayofwe
ek / dayofyear / days_in_month / daysinmonth / floor / freq / hour / is_leap
_year / is_month_end / is_month_start / is_quarter_end / is_quarter_start /
is_year_end / is_year_start / isocalendar / microsecond / minute / month / m
onth_name / nanosecond / normalize / quarter / round / second / strftime / t
ime / timetz / to_period / to_pydatetime / tz / tz_convert / tz_localize / u
nit / weekday / year /
```

For instance, one can get the hour of the day of a date-like column:

```
In [12]: df['Hour'] = df['date'].dt.hour
         df
```

Out[12]:

	date	A	B	Hour
0	2024-08-15 00:00:00	0	6	0
1	2024-08-15 02:00:00	1	7	2
2	2024-08-15 04:00:00	2	8	4
3	2024-08-15 06:00:00	3	9	6
4	2024-08-15 08:00:00	4	10	8
5	2024-08-15 10:00:00	5	11	10

Loading [MathJax]/extensions/Safe.js

TimeDelta

`pandas` can also handle date differences.

Hereafter, the first date is substracted from the other. This operation returns the durations from this initial date, for all elements of `df['date']` (because `df['date']` is chronologically sorted).

In [13]:
```
df['date'] - df.loc[0, 'date']
```

Out[13]:
```
0   0 days 00:00:00
1   0 days 02:00:00
2   0 days 04:00:00
3   0 days 06:00:00
4   0 days 08:00:00
5   0 days 10:00:00
Name: date, dtype: timedelta64[ns]
```

Loading [MathJax]/extensions/Safe.js