outatime

Release 3.1.0

Luca Spartera

Jun 28, 2022

TABLE OF CONTENTS

CHAPTER

ONE

ABOUT OUTATIME

The main goal of this framework is to simplify the collection of temporal data and related operations. It is based on the concepts of **TimeSeries**, as a collection of ordered records associated with a given day, and **Granularity**, to indicate a given time interval (e.g. daily, weekly, monthly, etc.).

1.1 Why outatime?

Outatime allows to carry out numerous operations of different types on time series, as for example:

- add or remove records
- easy access to records by date
- exclude records outside a determined range
- resampling of data
- querying data
- union and intersection
- batch splitting and data aggregation
- other

CHAPTER

TWO

INSTALLATION

To install it use:

```
$ pip install outatime
```

or download the last git version and use:

```
$ python setup.py install
```

CHAPTER

THREE

TUTORIAL

Each time series is structured as an ordered list of daily items, always arranged in ascending chronological order.

The object related to a single day (**TimeSeriesData**) contains two attributes:

- **day** - the reference date for that record
- **data** - an object that collects any information for that day

Before proceeding with the tutorial, it is essential to clarify the concept of granularity. A Granularity object defines the frequency of data contained in a time series.

The default supported granularity are:

- DailyGranularity
- WeeklyGranularity
- MonthlyGranularity
- QuarterlyGranularity
- YearlyGranularity

The granularity of a given time series is inferred completely independently from the distribution of the data in it. It is possible, however, when constructing the time series, to give as input a set of Granularity objects from which to search for the frequency closest to that of the input data.

3.1 Create a new time series

To create your first time series, you need to collect data in individual objects of type TimeSeriesData. A list of these objects can be fed to our TimeSeries class.

```
ts_data = [
    TimeSeriesData(
        day = datetime.date(2022, 6, 24),
        data = {
            "AAPL": 135.73,
            "MSFT": 251.81,
            "GOOGL": 2275.34
        }
    ),
    TimeSeriesData(
        day = datetime.date(2022, 6, 27),
        data = {
```

(continues on next page)

(continued from previous page)

```
            "AAPL": 142.16,
            "MSFT": 265.66,
            "GOOGL": 2337.92
        }
    )
]

ts = TimeSeries(ts_data)
```

You can add new data to your time series, that will keep the information in chronological order.

```
new_data = TimeSeriesData(
        day = datetime.date(2022, 6, 23),
        data = {
            "AAPL": 140.04,
            "MSFT": 249.65,
            "GOOGL": 2229.44
        }
    )

ts.append(new_data)
```

You can also update the time series with multiple new inputs.

```
new_data_list = [
    TimeSeriesData(
        day = datetime.date(2022, 6, 22),
        data = {
            "AAPL": 136.55,
            "MSFT": 246.07,
            "GOOGL": 2205.50
        }
    ),
    TimeSeriesData(
        day = datetime.date(2022, 6, 23),
        data = {
            "AAPL": 140.04,
            "MSFT": 249.65,
            "GOOGL": 2229.44
        }
    )
]

ts.update(new_data_list)
```

3.2 Retrieve data

There are different ways to retrieve data from your time series.

1. You can get a TimeSeriesData by its index in the TimeSeries object.

```
ts_data = ts[0]  # gets the first element in the time series
```

2. Alternatively you can search for an item by its date.

```
date_to_find = datetime.date(2022, 6, 23)
ts_data = ts.get(date_to_find)  # gets the element with the given date
```

3. In addition, a subset of the time series can be extracted using the **query** function. The user can specify filters in string format to be applied on the values of "day," "month," and "year." It is possible to create a new time series as output (by default) or overwrite the original one by setting the **inplace** parameter to **True**.

```
query_str = "month == 6 and day == 2022)
ts_subset = ts.query(query_str)  # extracts all data for the month of June for the year␣
↪2022
```

```
query_str = "month == 6 and day == 2022)
ts.query(query_str, inplace=True)  # extracts all data for the month of June for the␣
↪year 2022
```

3.3 Manage the time series

A single item can be removed from the time series by accessing it by date.

```
date_to_del = datetime.date(2022, 6, 23)
ts_data = ts.delete(date_to_del)  # removes the element with the given date
```

Given a range of dates, it is possible to remove all elements not included from the time series. To do this, the **cut** method is used, which allows you to create a new time series as output (by default) or overwrite the original one by setting the **inplace** parameter to **True**.

```
min_date = datetime.date(2022, 6, 23)
max_date = datetime.date(2022, 6, 25)
ts_data = ts.cut(min_date, max_date)  # removes items prior to June 23 and items after␣
↪June 25
```

```
min_date = datetime.date(2022, 6, 23)
max_date = datetime.date(2022, 6, 25)
ts.cut(min_date, max_date, inplace=True)  # removes items prior to June 23 and items␣
↪after June 25
```

It is possible to change the granularity of the time series through the resample function, which allows:

- **upsampling** - you move toward a lower granularity by increasing the number of elements.
- **downsampling** - you proceed toward a higher granularity by reducing the number of elements

You can define the function to be applied for generating the new data (e.g., the monthly value could correspond to the average of the daily values). Again, the **inplace** parameter can be set to **True** to overwrite the time series.

```
output_granularity = MonthlyGranularity()
resampled_ts = ts.resample(output_granularity, method=sum)
```

```
output_granularity = MonthlyGranularity()
ts.resample(output_granularity, method=sum, inplace=True)
```

3.4 Split the time series

Operations can be performed on the time series that involve dividing it into sections defined by a user-chosen granularity.

The **aggregate** method makes it possible to generate a new time series with reduced granularity by specifying which part of each interval to use. You can specify an aggregation method, as well as the target day of each interval on which to save the result.

For example, I might want to aggregate the data by month with the averaging function and save the result on day 15 of each month.

```
output_granularity = MonthlyGranularity()
aggr_ts = aggregate(ts, output_granularity, method=mean, store_day_of_batch=15)  #␣
↪saving results on the day 15 of month
```

You can create a new time series with only one specific day of each interval of a given granularity. To do this the function **pick_a_day** is used.

For example, I want to extract the first day of each month from the input data.

```
output_granularity = MonthlyGranularity()
new_ts = pick_a_day(ts, output_granularity, day_of_batch=0)
```

Similarly, with the **pick_a_weekday** method, it is possible to select a specific day of the week within a given range.

For example, I want to extract the first Thursday of each month of the input data.

```
output_granularity = MonthlyGranularity()
new_ts = pick_a_weekday(ts, output_granularity, weekday=4, day_of_batch=0)
```

The **split** method allows generating a list of TimeSeries objects, each obtained by separating the input time series into intervals of the given granularity.

For example, if I have a time series with daily granularity, I can get a list of time series, each containing data from a single month, by choosing monthly granularity.

```
output_granularity = MonthlyGranularity()
ts_list = split(ts, output_granularity)
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search