# Module 7 Part 2: Time Series

This module consists of 2 parts.

- **Part 1** - Forecasting with Pandas
- **Part 2** - Time Series

Each part is provided in a separate file. It is recommended that you follow the order of the files.

# Table of Contents



# Setup

To begin, we will import the following.

In [1]:
```
import numpy as np  # For computations
import pandas as pd  # For indexing our data

np.random.seed(12345)
import matplotlib.pyplot as plt  # for visualizing our data

plt.rc('figure', figsize=(10, 6))
PREVIOUS_MAX_ROWS = pd.options.display.max_rows
pd.options.display.max_rows = 20
np.set_printoptions(precision=4, suppress=True)

# Our temporal data types
from datetime import datetime
from datetime import timedelta

# For grabbing data sets
import requests
```

# Time Series

A **time series** is a set of observations at different points in time. A time series can have a **fixed frequency** of measured points (i.e. monthly), or have an **irregular frequency** of data points which are irregularly spaced in time (i.e. whenever data is available).

The time dimension in `pandas` objects can be marked and expressed in multiple ways.

| Marking Method | Example |
|---|---|
| Timestamps | December 13, 2017 at 11:22 EST |
| Fixed periods | monthly |
| Intervals | 2015-04-03 03:12 to 2015-04-14 11:11 |
| Elapsed time | 45 mins. 32:05 secs. |

**NOTE:** In `pandas`, time ranges can be utilized as an index. If a `Series` is created where the index is made from a list of `datetime` objects, the `Series` will become a time series. We will only focus on the time dimension used with `Series`, but the methods discussed in this module can easily be adjusted for `DataFrames` and `Panels`.

In [2]:
```
"""
Creating a datetime index.

start : str or datetime-like, optional (left bound for generating dates)
end : str or datetime-like, optional (right bound for generating dates)
periods : integer, optional (number of periods to generate)
freq : str or DateOffset, default 'D' (calendar daily)
    Frequency strings can have multiples, e.g. '5H'.
"""
timerange = pd.date_range('7/7/7', periods=7, freq='H') # Fixed frequency of hours
timerange
```

Out[2]:
```
DatetimeIndex(['2007-07-07 00:00:00', '2007-07-07 01:00:00',
               '2007-07-07 02:00:00', '2007-07-07 03:00:00',
               '2007-07-07 04:00:00', '2007-07-07 05:00:00',
               '2007-07-07 06:00:00'],
              dtype='datetime64[ns]', freq='H')
```

If a `Series` is created where the index is made from a list of `datetime` objects, the `Series` will become a time series.

Remember, the time dimension is simply a type of index which can be similarly applied on a `Series`, `DataFrame`, or `Panel`. This is because `DataFrame`s are created from `Series`, and `Panel`s from `DataFrame`s.

In [3]:
```
# Creating an arbitrary time series with random numbers
randomTimeSeries = pd.Series(np.random.randn(len(timerange)), index=timerange) + 1
```

```
randomTimeSeries
```

Out[3]:
```
2007-07-07 00:00:00    0.795292
2007-07-07 01:00:00    1.478943
2007-07-07 02:00:00    0.480561
2007-07-07 03:00:00    0.444270
2007-07-07 04:00:00    2.965781
2007-07-07 05:00:00    2.393406
2007-07-07 06:00:00    1.092908
Freq: H, dtype: float64
```

Unlike other index types, a unit of measure is now implicitly associated with the index. Additionally, time can be viewed at different granularities (i.e. days, weeks, months). Thus, it is important to know how to transform time scales in order to align data points.

In [4]:
```
# You can easily convert into different frequencies
# asfreq() provides us with easy conversion methods to express our index with diffe
"""
method : {‘backfill’/’bfill’, ‘pad’/’ffill’}, default None:
    Method to use for filling holes when switching frequencies (note this does not

‘pad’ / ‘ffill’: use LAST valid observation to fill (propagate) forward to next val
‘backfill’ / ‘bfill’: use NEXT valid observation to fill (propagate) backwards to l
"""
randomTimeSeries.asfreq(freq='30Min')
```

Out[4]:
```
2007-07-07 00:00:00    0.795292
2007-07-07 00:30:00         NaN
2007-07-07 01:00:00    1.478943
2007-07-07 01:30:00         NaN
2007-07-07 02:00:00    0.480561
2007-07-07 02:30:00         NaN
2007-07-07 03:00:00    0.444270
2007-07-07 03:30:00         NaN
2007-07-07 04:00:00    2.965781
2007-07-07 04:30:00         NaN
2007-07-07 05:00:00    2.393406
2007-07-07 05:30:00         NaN
2007-07-07 06:00:00    1.092908
Freq: 30T, dtype: float64
```

In [5]:
```
# You can add value repetition forward (method= 'pad')
# You can also try to fill backwards using method= 'bfill'
# We can get a timeseries with or without value repetition forward or backward
# NOTE: Use case is for when our values are continuous and not discrete bursts
randomTimeSeries.asfreq(freq='30Min', method='pad')
```

Out[5]:
```
2007-07-07 00:00:00    0.795292
2007-07-07 00:30:00    0.795292
2007-07-07 01:00:00    1.478943
2007-07-07 01:30:00    1.478943
2007-07-07 02:00:00    0.480561
2007-07-07 02:30:00    0.480561
2007-07-07 03:00:00    0.444270
2007-07-07 03:30:00    0.444270
2007-07-07 04:00:00    2.965781
2007-07-07 04:30:00    2.965781
2007-07-07 05:00:00    2.393406
2007-07-07 05:30:00    2.393406
2007-07-07 06:00:00    1.092908
Freq: 30T, dtype: float64
```

Arithmetic between differently-indexed time series automatically align on dates. The indexing, selection, and subsets work the way we've seen for `DataFrames` and `Series`, since time ranges conform to a type of `pandas` index with a few extra methods. Likewise, duplicate index timestamps are allowed — similar to regular indexes.

In [6]:
```
# For example, we can apply simple operations to the time series, like so below.
# Time Series behave similar to numpy ndarrays, series, and data frames
# (i.e. list-like comprehensions when selecting and transforming).
2 * randomTimeSeries
```

Out[6]:
```
2007-07-07 00:00:00    1.590585
2007-07-07 01:00:00    2.957887
2007-07-07 02:00:00    0.961123
2007-07-07 03:00:00    0.888539
2007-07-07 04:00:00    5.931561
2007-07-07 05:00:00    4.786812
2007-07-07 06:00:00    2.185816
Freq: H, dtype: float64
```

In [7]:
```
# But... changing frequencies to smaller periods adds a lot of `NaN` values (when w
# So be mindful. The values might not transform the way you expect
# (i.e., `NaN + 3 = NaN` and not `3`)
randomTimeSeries.asfreq(freq='30Min', method='pad') + randomTimeSeries.asfreq(freq=
```

Out[7]:
```
2007-07-07 00:00:00    1.590585
2007-07-07 00:30:00         NaN
2007-07-07 00:50:00         NaN
2007-07-07 01:00:00         NaN
2007-07-07 01:30:00         NaN
2007-07-07 01:40:00         NaN
2007-07-07 02:00:00         NaN
2007-07-07 02:30:00    0.961123
2007-07-07 03:00:00         NaN
2007-07-07 03:20:00         NaN
2007-07-07 03:30:00         NaN
2007-07-07 04:00:00         NaN
2007-07-07 04:10:00         NaN
2007-07-07 04:30:00         NaN
2007-07-07 05:00:00    4.786812
2007-07-07 05:30:00         NaN
2007-07-07 05:50:00         NaN
2007-07-07 06:00:00         NaN
dtype: float64
```

In [8]:
```
# Finally, index entries can have duplication, just like any other index
newTimerange = timerange.append(timerange)
newTimeSeries = pd.Series(np.random.randn(len(newTimerange)), index=newTimerange)
newTimeSeries
```

Out[8]:
```
2007-07-07 00:00:00    0.281746
2007-07-07 01:00:00    0.769023
2007-07-07 02:00:00    1.246435
2007-07-07 03:00:00    1.007189
2007-07-07 04:00:00   -1.296221
2007-07-07 05:00:00    0.274992
2007-07-07 06:00:00    0.228913
2007-07-07 00:00:00    1.352917
2007-07-07 01:00:00    0.886429
2007-07-07 02:00:00   -2.001637
2007-07-07 03:00:00   -0.371843
2007-07-07 04:00:00    1.669025
2007-07-07 05:00:00   -0.438570
2007-07-07 06:00:00   -0.539741
dtype: float64
```

However, unlike regular indexes, time ranges can be relabeled by shifting dates (i.e. adding a time offset) or resampling (i.e. reconstructing a time series from itself).

In [9]:
```
# Shifting time values with timedelta, without changing the index structure.
# Pay attention to `freq` value. All values are shifted. The frequency is still the
timerange + pd.Timedelta("3Min")
```

Out[9]:
```
DatetimeIndex(['2007-07-07 00:03:00', '2007-07-07 01:03:00',
               '2007-07-07 02:03:00', '2007-07-07 03:03:00',
               '2007-07-07 04:03:00', '2007-07-07 05:03:00',
               '2007-07-07 06:03:00'],
              dtype='datetime64[ns]', freq='H')
```

Additionally, both `Series` and `DataFrame`s have a `shift()` method which shifts *data* without changing the index. The shift is specified as multiples of the frequency. Positive values shift past values forward.

In [10]:
```
# With shift(), we can shift the `TimeSeries` (not the time range)
{"beforeShift":randomTimeSeries, "afterShift" : randomTimeSeries.shift(1),
 "factorTrend" : randomTimeSeries / randomTimeSeries.shift(1)}
```

Out[10]:
```
{'beforeShift': 2007-07-07 00:00:00    0.795292
 2007-07-07 01:00:00    1.478943
 2007-07-07 02:00:00    0.480561
 2007-07-07 03:00:00    0.444270
 2007-07-07 04:00:00    2.965781
 2007-07-07 05:00:00    2.393406
 2007-07-07 06:00:00    1.092908
 Freq: H, dtype: float64, 'afterShift': 2007-07-07 00:00:00         NaN
 2007-07-07 01:00:00    0.795292
 2007-07-07 02:00:00    1.478943
 2007-07-07 03:00:00    0.480561
 2007-07-07 04:00:00    0.444270
 2007-07-07 05:00:00    2.965781
 2007-07-07 06:00:00    2.393406
 Freq: H, dtype: float64, 'factorTrend': 2007-07-07 00:00:00         NaN
 2007-07-07 01:00:00    1.859622
 2007-07-07 02:00:00    0.324936
 2007-07-07 03:00:00    0.924481
 2007-07-07 04:00:00    6.675631
 2007-07-07 05:00:00    0.807007
 2007-07-07 06:00:00    0.456633
 Freq: H, dtype: float64}
```

For this module, we will only consider time series that are observed at *regular intervals of time* (e.g. hourly, daily, weekly, monthly, quarterly, annually). The base frequency identifiers used to make time ranges in `pandas` have a lot of built-in knowledge about business calendars, allowing indexes to take into account business hours, holidays, weekends, etc. Inspect them carefully, as selecting the right frequency can avoid having to deal with special calendar cases, such as leap years.

*Irregularly* spaced time series can also occur, but are beyond our scope of discussion. To avoid irregularity, we can convert an irregular time series to a fixed frequency where newly created observations for times without data will get values of `NaN`.

In [11]:
```
# Here we are resampling an irregular timeseries to force regularity.
# **NOTE:** The aggregate sum() smooths and regularizes the series.
# **NOTE:** The duplicates were also affected by the aggregates
{
    "original":newTimeSeries,
    "resampled":newTimeSeries.resample("30Min"), # a function, not a Series
```

```
    "aggregatedResample":newTimeSeries.resample("30Min").sum() # aggregate to compu
}
```

Out[11]:
```
{'original': 2007-07-07 00:00:00    0.281746
 2007-07-07 01:00:00    0.769023
 2007-07-07 02:00:00    1.246435
 2007-07-07 03:00:00    1.007189
 2007-07-07 04:00:00   -1.296221
 2007-07-07 05:00:00    0.274992
 2007-07-07 06:00:00    0.228913
 2007-07-07 00:00:00    1.352917
 2007-07-07 01:00:00    0.886429
 2007-07-07 02:00:00   -2.001637
 2007-07-07 03:00:00   -0.371843
 2007-07-07 04:00:00    1.669025
 2007-07-07 05:00:00   -0.438570
 2007-07-07 06:00:00   -0.539741
 dtype: float64,
 'resampled': DatetimeIndexResampler [freq=<30 * Minutes>, axis=0, closed=left, la
bel=left, convention=start, base=0],
 'aggregatedResample': 2007-07-07 00:00:00    1.634663
 2007-07-07 00:30:00    0.000000
 2007-07-07 01:00:00    1.655452
 2007-07-07 01:30:00    0.000000
 2007-07-07 02:00:00   -0.755203
 2007-07-07 02:30:00    0.000000
 2007-07-07 03:00:00    0.635347
 2007-07-07 03:30:00    0.000000
 2007-07-07 04:00:00    0.372804
 2007-07-07 04:30:00    0.000000
 2007-07-07 05:00:00   -0.163578
 2007-07-07 05:30:00    0.000000
 2007-07-07 06:00:00   -0.310829
 Freq: 30T, dtype: float64}
```

**End of Module**

You have reached the end of this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

When you are comfortable with the content, and have practiced to your satisfaction, you may proceed to any related assignments, and to the next module.

# Additional Resources


- Brodersen, H., Gallusser, F., Kay, H. & Keohler, J. (2015). *Inferring causal impact using bayesian structural time-series models*'', Annals of Applied Statistics, vol. 9, number , pp. 247-274. online

- Broderson, H. & Kay, H. (2014). *Causalimpact: a new open-source package for estimating causal effects in time series | google open source blog*. online
- Hilpisch, Y. (2014). *Python for finance: analyze big financial data*.
- Natrella, M. (2013). ``*Nist/sematech e-handbook of statistical methods*'', October 2013. online
- Sargent, T. & Stachurski, J. (2017). ``*Quantitative economics*'', 2017. online
- Shumway, R.H. & Stoffer, D.S. (2017). *Time series analysis using the r statistical package*. online
- Srivastava, T. (2015). *A Complete Tutorial on Time Series Modeling in R*. online
- Ulrich, J. (2018). ``*Foss trading*''. online


# References


Hyndman, R.J. & Athanasopoulos, G. (2018). *Forecasting: principles and practice, 2nd Ed.*. online

Pandas Contributors (2018). Time series / date functionality — pandas 0.23.3 documentation. online


In [ ]: