# Chapter 3
# Time-aware DataFrames

## 3.1 Overview

Time series have two properties: a value for the time and a value that was measured at that time. Often these are displayed as two separate variables, i.e. a column for the time and a column for the “value”. However the philosophy of **pandas** is that these two belong together hence the need for a DateTimeIndex. In this section we will explore the advantages this index provides and show how it supports other functionality.

## 3.2 The DateTimeIndex

By default the index of a DataFrame is numeric, usually the equivalence of a row number. But the index is meant to be much more than just a row number. It is also used for aligning records after certain operations and selecting records through slicing. This doesn’t just apply to a numeric index but also to any other type of index and is the reason why having a DateTimeIndex is so useful.

There are two ways to create a DataFrame with a DateTimeIndex: After creation of the DataFrame and during creation of the DataFrame:

```
>>> df = pd.DataFrame({"Value":range(5)})
>>> my_dates = pd.date_range("01-01-2018", periods=5, freq="M")
>>> df.index = my_dates
>>> df

            Value
2018-01-31      0
2018-02-28      1
2018-03-31      2
2018-04-30      3
2018-05-31      4

>>> df = pd.DataFrame({"Value":range(5)}, index=my_dates)
>>> df

            Value
2018-01-31      0
2018-02-28      1
2018-03-31      2
2018-04-30      3
2018-05-31      4
```

If your dates are included in your data you can also use the `set_index` function to set it as the index. You can even use the `read_csv` function to read in the date column as a date (set `parse_dates` to *True*) and then use that as the index (set `index_col` to the name of the date column, e.g. “Date”)

## 3.3 Selecting Records

With the index set we can now show one the great capabilities of having a DateTimeIndex, namely slicing using time.

```
>>> # select rows by a specific value
>>> df.loc["2018-01-31"]

Value    0
Name: 2018-01-31 00:00:00, dtype: int64

>>> # select multiple rows using a range of values
>>> df["2018-01-01":"2018-03-01"]

            Value
2018-01-31      0
2018-02-28      1

>>> # select multiple rows using a hierarchically higher value
>>> df["2018-01"]

            Value
2018-01-31      0

>>> df["2018"]

            Value
2018-01-31      0
2018-02-28      1
2018-03-31      2
2018-04-30      3
2018-05-31      4
```

So we can select by specifying a range of date values just like we would specify a range of numbers. **pandas** conveniently converts the strings to datetime objects for us. Additionally we can select using a hierarchically higher level such as month or year to select all values from that level.

●

1. Load in `dji.csv` and ensure the Date column is set as the index.
2. Select all values from the summer (between May and August).
3. Is there anything unusual about the dates in this dataset?|

## 3.4 DateTimeIndex-Aware Functions

Beyond the slicing capability, a DateTimeIndex also opens a host of other time series related functions.

### *3.4.1 Resampling*

To analyse a time series it helps if the frequency of the time axis is consistent. A time series with a fixed number of observations per unit of time is much easier to analyse than a time series that has a random number of observations. **pandas**'s `resample` function helps us convert one into the other. We can, for example, change a time series with values at the minute level to a time series with values at the hour level.

The `resample` function itself returns an object of type DateTimeIndexResampler. To obtain the new values we must apply an aggregation and we can pick any of the aggregation functions we would normally use with a groupby operation (e.g. mean, max, median, etc.).

```
>>> df = pd.DataFrame({"Value":[10, 20, 40, 80, 160, 320]},
index=pd.date_range("01-01-2018", periods=6, freq="30T"))
>>> df

            Value
2018-01...     10
2018-01...     20
2018-01...     40
2018-01...     80
2018-01...    160
2018-01...    320

>>> hourly_df = df.resample("H").sum()
>>> hourly_df

            Value
2018-01...     30
2018-01...    120
2018-01...    480
```

The above example is an example of down-sampling. `resample` also allows us to go the other direction and up-sample the time frequency. By default `resample` will leave any newly created values as missing values (*NA*). However we can specify how we want to impute these missing values using one of the fill functions (`pad`, `bfill`, `ffill` or a custom function).

```
>>> hourly_df.resample("30T").asfreq()

            Value
2018-01...   30.0
2018-01...    NaN
2018-01...  120.0
2018-01...    NaN
2018-01...  480.0

>>> hourly_df.resample("30T").ffill()

            Value
2018-01...     30
2018-01...     30
2018-01...    120
2018-01...    120
2018-01...    480
```

If instead of aggregating the data as part of the down-sampling we wanted to select values we can use the `as.freq` function. This function accepts the same arguments as `resample`.

### 3.4.2 Shift

Suppose we had monthly sales data and we wanted to know how much our sales had increased or decreased compared to the previous month. This is a problem we can solve with the `diff` function:

```
>>> sales_data = pd.DataFrame({"Sales":[10, 20, 40, 80]},
index=pd.date_range("01-07-2018", periods=4, freq="M"))
>>> sales_data.diff()

            Sales
2018-01-31    NaN
2018-02-28   10.0
2018-03-31   20.0
2018-04-30   40.0
```

The `diff` function calculates the difference between a value and another value within the same column. You can control what this other value is through the `periods` parameter. A positive number for `periods` will take values from before the current value and a negative number will take values from after the current value. For example, a value of -2 will take the difference between the current value and the value that is two records after.

```
>>> sales_data.diff(periods=-2)

            Sales
2018-01-31  -30.0
2018-02-28  -60.0
2018-03-31    NaN
2018-04-30    NaN
```

● You can also take the difference between columns by setting the `axis` argument to “columns” or 1.

The `diff` function only looks at the number of records before or after the current record. As such it does not actually take into account the time dimension. The `diff` function is also restricted to only taking the difference between values. To compare values from different points in time in another way we can use the `shift` function. The `shift` function shifts the data a number of periods according to the frequency of the data or a user-specified frequency.

```
>>> sales_data["previous_month"] = sales_data.shift(1)
>>> sales_data

            Sales  previous_month
2018-01-31     10             NaN
2018-02-28     20            10.0
2018-03-31     40            20.0
2018-04-30     80            40.0
```

### *3.4.3 Rolling windows*

One of the more common operations performed on time series data is to aggregate data from a certain period repeatedly. This period is also referred to as a sliding or rolling window. This functionality is supported by **pandas** through the `rolling` function. Similar to the `resample` function, `rolling` returns an object of type Rolling upon which we can apply our aggregation functions.

```
>>> df = pd.DataFrame({"Value":[10, 20, 40, 80]*10},
index=pd.date_range("01-01-2018", periods=40, freq="D"))
>>> df.rolling(window=7).mean()[4:9]

                Value
2018-01-05        NaN
2018-01-06        NaN
2018-01-07  31.428571
2018-01-08  41.428571
2018-01-09  40.000000
```

The above example calculates the moving average for a period of 7 days. By default, `rolling` does not provide a result until it has at least a number of values that is equal to the size of the window (in this case 7). We can control this behaviour using the `min_periods` argument.

```
>>> df.rolling(window=7, min_periods=6).mean()[4:9]

                Value
2018-01-05        NaN
2018-01-06  30.000000
2018-01-07  31.428571
2018-01-08  41.428571
2018-01-09  40.000000
```

**pandas** also allows you to use a custom function and pass it on to `apply`.

```
>>> # calculate the range (passing raw=False to silence warning)
>>> df.rolling(window=7).apply(lambda window: window.max()-
window.min(), raw=False)[4:9]

            Value
2018-01-05    NaN
2018-01-06    NaN
2018-01-07   70.0
2018-01-08   70.0
2018-01-09   70.0
```

1. Using the previously loaded `dji.csv` data, up-sample the data to include the missing dates. Select a suitable fill value.
2. Calculating a moving average using the `rolling` function over a suitable period.
3. Down-sample the data by taking the monthly average and then calculate the month over month change