- location: the id of the sensor, either *FR04014*, *BETR801* or *London Westminster*
- parameter: the parameter measured by the sensor, either NO_2 or Particulate matter
- value: the measured value
- unit: the unit of the measured parameter, in this case 'µg/m^3'

and the index of the `DataFrame` is `datetime`, the datetime of the measurement.

Note: The air-quality data is provided in a so-called *long format* data representation with each observation on a separate row and each variable a separate column of the data table. The long/narrow format is also known as the tidy data format.

```
In [4]: air_quality = pd.read_csv("data/air_quality_long.csv",
   ...:                           index_col="date.utc", parse_dates=True)
   ...:

In [5]: air_quality.head()
Out[5]:
                                city country location parameter  value   unit
date.utc
2019-06-18 06:00:00+00:00  Antwerpen      BE  BETR801      pm25   18.0  µg/m³
2019-06-17 08:00:00+00:00  Antwerpen      BE  BETR801      pm25    6.5  µg/m³
2019-06-17 07:00:00+00:00  Antwerpen      BE  BETR801      pm25   18.5  µg/m³
2019-06-17 06:00:00+00:00  Antwerpen      BE  BETR801      pm25   16.0  µg/m³
2019-06-17 05:00:00+00:00  Antwerpen      BE  BETR801      pm25    7.5  µg/m³
```

How to reshape the layout of tables?

Sort table rows

I want to sort the titanic data according to the age of the passengers.

```
In [6]: titanic.sort_values(by="Age").head()
Out[6]:
     PassengerId  Survived  Pclass                             Name     Sex   Age ␣
↪SibSp  Parch  Ticket     Fare Cabin Embarked
803          804         1       3  Thomas, Master. Assad Alexander    male  0.42    ␣
↪ 0      1    2625   8.5167   NaN        C
755          756         1       2        Hamalainen, Master. Viljo    male  0.67    ␣
↪ 1      1  250649  14.5000   NaN        S
644          645         1       3           Baclini, Miss. Eugenie  female  0.75    ␣
↪ 2      1    2666  19.2583   NaN        C
469          470         1       3    Baclini, Miss. Helene Barbara  female  0.75    ␣
↪ 2      1    2666  19.2583   NaN        C
78            79         1       2    Caldwell, Master. Alden Gates    male  0.83    ␣
↪ 0      2  248738  29.0000   NaN        S
```

I want to sort the titanic data according to the cabin class and age in descending order.

```
In [7]: titanic.sort_values(by=['Pclass', 'Age'], ascending=False).head()
Out[7]:
     PassengerId  Survived  Pclass                       Name     Sex   Age  SibSp ␣
↪Parch  Ticket     Fare Cabin Embarked
```

(continues on next page)

(continued from previous page)

```
851          852         0        3        Svensson, Mr. Johan    male  74.0       0  ␣
↪  0  347060  7.7750    NaN        S
116          117         0        3       Connors, Mr. Patrick    male  70.5       0  ␣
↪  0  370369  7.7500    NaN        Q
280          281         0        3           Duane, Mr. Frank    male  65.0       0  ␣
↪  0  336439  7.7500    NaN        Q
483          484         1        3     Turkula, Mrs. (Hedwig)  female  63.0       0  ␣
↪  0    4134  9.5875    NaN        S
326          327         0        3  Nysveen, Mr. Johan Hansen    male  61.0       0  ␣
↪  0  345364  6.2375    NaN        S
```

With `Series.sort_values()`, the rows in the table are sorted according to the defined column(s). The index will follow the row order.

More details about sorting of tables is provided in the using guide section on *sorting data*.

Long to wide table format

Let's use a small subset of the air quality data set. We focus on NO_2 data and only use the first two measurements of each location (i.e. the head of each group). The subset of data will be called `no2_subset`

```
# filter for no2 data only
In [8]: no2 = air_quality[air_quality["parameter"] == "no2"]
```

```
# use 2 measurements (head) for each location (groupby)
In [9]: no2_subset = no2.sort_index().groupby(["location"]).head(2)

In [10]: no2_subset
Out[10]:
                                city country            location parameter  value  ␣
↪unit
date.utc                                                                             ␣
↪
2019-04-09 01:00:00+00:00  Antwerpen      BE             BETR801       no2   22.5  µg/
↪m³
2019-04-09 01:00:00+00:00      Paris      FR             FR04014       no2   24.4  µg/
↪m³
2019-04-09 02:00:00+00:00     London      GB  London Westminster       no2   67.0  µg/
↪m³
2019-04-09 02:00:00+00:00  Antwerpen      BE             BETR801       no2   53.5  µg/
↪m³
2019-04-09 02:00:00+00:00      Paris      FR             FR04014       no2   27.4  µg/
↪m³
2019-04-09 03:00:00+00:00     London      GB  London Westminster       no2   67.0  µg/
↪m³
```

I want the values for the three stations as separate columns next to each other

```
In [11]: no2_subset.pivot(columns="location", values="value")
Out[11]:
location                   BETR801  FR04014  London Westminster
date.utc
2019-04-09 01:00:00+00:00     22.5     24.4                 NaN
```

(continues on next page)

(continued from previous page)

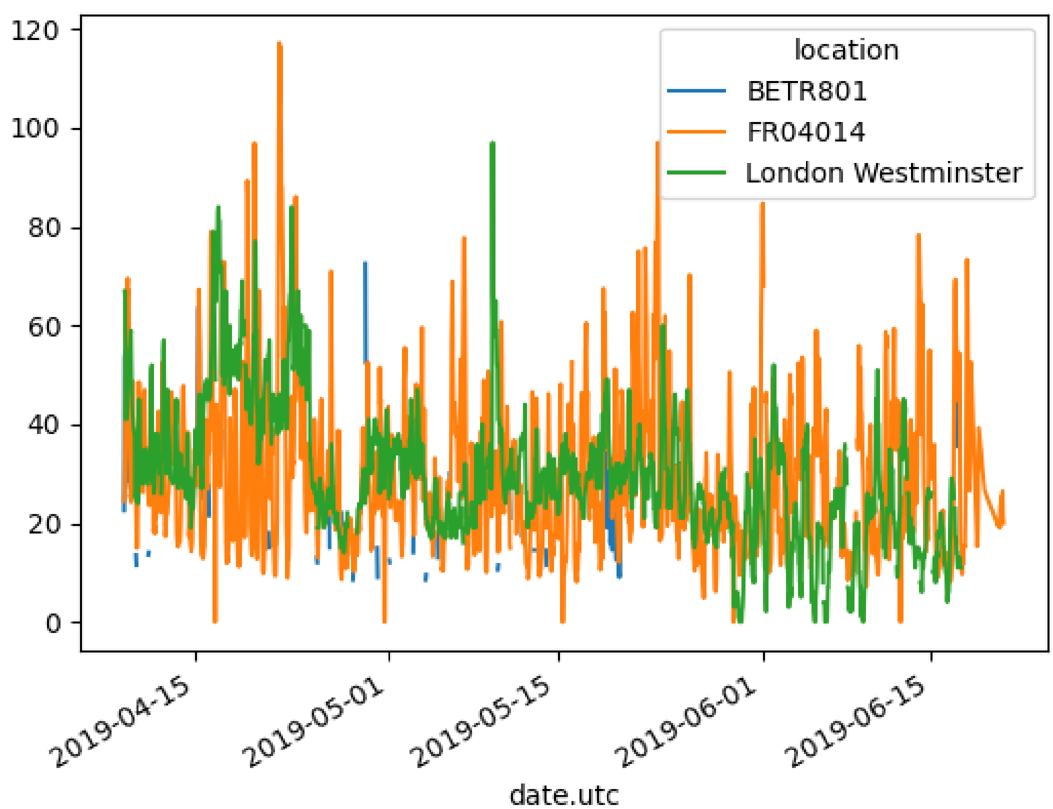
```
2019-04-09 02:00:00+00:00      53.5      27.4                    67.0
2019-04-09 03:00:00+00:00       NaN       NaN                    67.0
```

The `pivot_table()` function is purely reshaping of the data: a single value for each index/column combination is required.

As pandas support plotting of multiple columns (see *plotting tutorial*) out of the box, the conversion from *long* to *wide* table format enables the plotting of the different time series at the same time:

```
In [12]: no2.head()
Out[12]:
                            city country location parameter  value   unit
date.utc
2019-06-21 00:00:00+00:00  Paris      FR  FR04014       no2   20.0  µg/m³
2019-06-20 23:00:00+00:00  Paris      FR  FR04014       no2   21.8  µg/m³
2019-06-20 22:00:00+00:00  Paris      FR  FR04014       no2   26.5  µg/m³
2019-06-20 21:00:00+00:00  Paris      FR  FR04014       no2   24.9  µg/m³
2019-06-20 20:00:00+00:00  Paris      FR  FR04014       no2   21.4  µg/m³
```

```
In [13]: no2.pivot(columns="location", values="value").plot()
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7f534422df50>
```



Note: When the `index` parameter is not defined, the existing index (row labels) is used.

For more information about *pivot()*, see the user guide section on *pivoting DataFrame objects*.

Pivot table

I want the mean concentrations for NO_2 and $PM_{2.5}$ in each of the stations in table form

```
In [14]: air_quality.pivot_table(values="value", index="location",
   ....:                          columns="parameter", aggfunc="mean")
   ....:
Out[14]:
parameter                 no2       pm25
location
BETR801             26.950920  23.169492
FR04014             29.374284        NaN
London Westminster  29.740050  13.443568
```

In the case of *pivot()*, the data is only rearranged. When multiple values need to be aggregated (in this specific case, the values on different time steps) *pivot_table()* can be used, providing an aggregation function (e.g. mean) on how to combine these values.

Pivot table is a well known concept in spreadsheet software. When interested in summary columns for each variable separately as well, put the `margin` parameter to `True`:

```
In [15]: air_quality.pivot_table(values="value", index="location",
   ....:                          columns="parameter", aggfunc="mean",
   ....:                          margins=True)
   ....:
Out[15]:
parameter                 no2       pm25        All
location
BETR801             26.950920  23.169492  24.982353
FR04014             29.374284        NaN  29.374284
London Westminster  29.740050  13.443568  21.491708
All                 29.430316  14.386849  24.222743
```

For more information about *pivot_table()*, see the user guide section on *pivot tables*.

Note: If case you are wondering, *pivot_table()* is indeed directly linked to *groupby()*. The same result can be derived by grouping on both `parameter` and `location`:

```
air_quality.groupby(["parameter", "location"]).mean()
```

Have a look at *groupby()* in combination with *unstack()* at the user guide section on *combining stats and groupby*.

Wide to long format

Starting again from the wide format table created in the previous section:

```
In [16]: no2_pivoted = no2.pivot(columns="location", values="value").reset_index()

In [17]: no2_pivoted.head()
Out[17]:
location                  date.utc  BETR801  FR04014  London Westminster
0        2019-04-09 01:00:00+00:00     22.5     24.4                 NaN
1        2019-04-09 02:00:00+00:00     53.5     27.4                67.0
2        2019-04-09 03:00:00+00:00     54.5     34.2                67.0
3        2019-04-09 04:00:00+00:00     34.5     48.5                41.0
4        2019-04-09 05:00:00+00:00     46.5     59.5                41.0
```

I want to collect all air quality NO_2 measurements in a single column (long format)

```
In [18]: no_2 = no2_pivoted.melt(id_vars="date.utc")

In [19]: no_2.head()
Out[19]:
                   date.utc location  value
0 2019-04-09 01:00:00+00:00  BETR801   22.5
1 2019-04-09 02:00:00+00:00  BETR801   53.5
2 2019-04-09 03:00:00+00:00  BETR801   54.5
3 2019-04-09 04:00:00+00:00  BETR801   34.5
4 2019-04-09 05:00:00+00:00  BETR801   46.5
```

The *pandas.melt()* method on a `DataFrame` converts the data table from wide format to long format. The column headers become the variable names in a newly created column.

The solution is the short version on how to apply *pandas.melt()*. The method will *melt* all columns NOT mentioned in `id_vars` together into two columns: A columns with the column header names and a column with the values itself. The latter column gets by default the name `value`.

The *pandas.melt()* method can be defined in more detail:

```
In [20]: no_2 = no2_pivoted.melt(id_vars="date.utc",
   ....:                         value_vars=["BETR801",
   ....:                                     "FR04014",
   ....:                                     "London Westminster"],
   ....:                         value_name="NO_2",
   ....:                         var_name="id_location")
   ....:

In [21]: no_2.head()
Out[21]:
                   date.utc id_location  NO_2
0 2019-04-09 01:00:00+00:00     BETR801  22.5
1 2019-04-09 02:00:00+00:00     BETR801  53.5
2 2019-04-09 03:00:00+00:00     BETR801  54.5
3 2019-04-09 04:00:00+00:00     BETR801  34.5
4 2019-04-09 05:00:00+00:00     BETR801  46.5
```

The result in the same, but in more detail defined:

- `value_vars` defines explicitly which columns to *melt* together

- `value_name` provides a custom column name for the values column instead of the default columns name `value`
- `var_name` provides a custom column name for the columns collecting the column header names. Otherwise it takes the index name or a default `variable`

Hence, the arguments `value_name` and `var_name` are just user-defined names for the two generated columns. The columns to melt are defined by `id_vars` and `value_vars`.

Conversion from wide to long format with `pandas.melt()` is explained in the user guide section on *reshaping by melt*.

- Sorting by one or more columns is supported by `sort_values`
- The `pivot` function is purely restructering of the data, `pivot_table` supports aggregations
- The reverse of `pivot` (long to wide format) is `melt` (wide to long format)

A full overview is available in the user guide on the pages about *reshaping and pivoting*.

```
In [1]: import pandas as pd
```

For this tutorial, air quality data about NO_2 is used, made available by openaq and downloaded using the py-openaq package.

The `air_quality_no2_long.csv` data set provides NO_2 values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [2]: air_quality_no2 = pd.read_csv("data/air_quality_no2_long.csv",
   ...:                                parse_dates=True)
   ...:

In [3]: air_quality_no2 = air_quality_no2[["date.utc", "location",
   ...:                                     "parameter", "value"]]
   ...:

In [4]: air_quality_no2.head()
Out[4]:
                    date.utc location parameter  value
0  2019-06-21 00:00:00+00:00  FR04014       no2   20.0
1  2019-06-20 23:00:00+00:00  FR04014       no2   21.8
2  2019-06-20 22:00:00+00:00  FR04014       no2   26.5
3  2019-06-20 21:00:00+00:00  FR04014       no2   24.9
4  2019-06-20 20:00:00+00:00  FR04014       no2   21.4
```

For this tutorial, air quality data about Particulate matter less than 2.5 micrometers is used, made available by openaq and downloaded using the py-openaq package.

The `air_quality_pm25_long.csv` data set provides PM_{25} values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [5]: air_quality_pm25 = pd.read_csv("data/air_quality_pm25_long.csv",
   ...:                                 parse_dates=True)
   ...:

In [6]: air_quality_pm25 = air_quality_pm25[["date.utc", "location",
   ...:                                       "parameter", "value"]]
   ...:

In [7]: air_quality_pm25.head()
```

(continues on next page)

(continued from previous page)

```
Out[7]:
                    date.utc location parameter  value
0  2019-06-18 06:00:00+00:00  BETR801      pm25   18.0
1  2019-06-17 08:00:00+00:00  BETR801      pm25    6.5
2  2019-06-17 07:00:00+00:00  BETR801      pm25   18.5
3  2019-06-17 06:00:00+00:00  BETR801      pm25   16.0
4  2019-06-17 05:00:00+00:00  BETR801      pm25    7.5
```

How to combine data from multiple tables?

Concatenating objects

I want to combine the measurements of NO_2 and PM_{25}, two tables with a similar structure, in a single table

```
In [8]: air_quality = pd.concat([air_quality_pm25, air_quality_no2], axis=0)

In [9]: air_quality.head()
Out[9]:
                    date.utc location parameter  value
0  2019-06-18 06:00:00+00:00  BETR801      pm25   18.0
1  2019-06-17 08:00:00+00:00  BETR801      pm25    6.5
2  2019-06-17 07:00:00+00:00  BETR801      pm25   18.5
3  2019-06-17 06:00:00+00:00  BETR801      pm25   16.0
4  2019-06-17 05:00:00+00:00  BETR801      pm25    7.5
```

The `concat()` function performs concatenation operations of multiple tables along one of the axis (row-wise or column-wise).

By default concatenation is along axis 0, so the resulting table combines the rows of the input tables. Let's check the shape of the original and the concatenated tables to verify the operation:

```
In [10]: print('Shape of the `air_quality_pm25` table: ', air_quality_pm25.shape)
Shape of the `air_quality_pm25` table:  (1110, 4)

In [11]: print('Shape of the `air_quality_no2` table: ', air_quality_no2.shape)
Shape of the `air_quality_no2` table:  (2068, 4)

In [12]: print('Shape of the resulting `air_quality` table: ', air_quality.shape)
Shape of the resulting `air_quality` table:  (3178, 4)
```

Hence, the resulting table has 3178 = 1110 + 2068 rows.

Note: The **axis** argument will return in a number of pandas methods that can be applied **along an axis**. A `DataFrame` has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1). Most operations like concatenation or summary statistics are by default across rows (axis 0), but can be applied across columns as well.

Sorting the table on the datetime information illustrates also the combination of both tables, with the `parameter` column defining the origin of the table (either `no2` from table `air_quality_no2` or `pm25` from table `air_quality_pm25`):

```
In [13]: air_quality = air_quality.sort_values("date.utc")

In [14]: air_quality.head()
Out[14]:
                       date.utc            location parameter  value
2067  2019-05-07 01:00:00+00:00  London Westminster       no2   23.0
1003  2019-05-07 01:00:00+00:00             FR04014       no2   25.0
100   2019-05-07 01:00:00+00:00             BETR801      pm25   12.5
1098  2019-05-07 01:00:00+00:00             BETR801       no2   50.5
1109  2019-05-07 01:00:00+00:00  London Westminster      pm25    8.0
```

In this specific example, the `parameter` column provided by the data ensures that each of the original tables can be identified. This is not always the case. the `concat` function provides a convenient solution with the `keys` argument, adding an additional (hierarchical) row index. For example:

```
In [15]: air_quality_ = pd.concat([air_quality_pm25, air_quality_no2],
   ....:                          keys=["PM25", "NO2"])
   ....:
```

```
In [16]: air_quality_.head()
Out[16]:
                         date.utc location parameter  value
PM25 0  2019-06-18 06:00:00+00:00  BETR801      pm25   18.0
     1  2019-06-17 08:00:00+00:00  BETR801      pm25    6.5
     2  2019-06-17 07:00:00+00:00  BETR801      pm25   18.5
     3  2019-06-17 06:00:00+00:00  BETR801      pm25   16.0
     4  2019-06-17 05:00:00+00:00  BETR801      pm25    7.5
```

Note: The existence of multiple row/column indices at the same time has not been mentioned within these tutorials. *Hierarchical indexing* or *MultiIndex* is an advanced and powerfull pandas feature to analyze higher dimensional data.

Multi-indexing is out of scope for this pandas introduction. For the moment, remember that the function `reset_index` can be used to convert any level of an index to a column, e.g. `air_quality.reset_index(level=0)`

Feel free to dive into the world of multi-indexing at the user guide section on *advanced indexing*.

More options on table concatenation (row and column wise) and how `concat` can be used to define the logic (union or intersection) of the indexes on the other axes is provided at the section on *object concatenation*.

Join tables using a common identifier

Add the station coordinates, provided by the stations metadata table, to the corresponding rows in the measurements table.

Warning: The air quality measurement station coordinates are stored in a data file `air_quality_stations.csv`, downloaded using the py-openaq package.

```
In [17]: stations_coord = pd.read_csv("data/air_quality_stations.csv")
```

(continues on next page)

(continued from previous page)

```
In [18]: stations_coord.head()
Out[18]:
  location  coordinates.latitude  coordinates.longitude
0  BELAL01              51.23619                4.38522
1  BELHB23              51.17030                4.34100
2  BELLD01              51.10998                5.00486
3  BELLD02              51.12038                5.02155
4  BELR833              51.32766                4.36226
```

Note: The stations used in this example (FR04014, BETR801 and London Westminster) are just three entries enlisted in the metadata table. We only want to add the coordinates of these three to the measurements table, each on the corresponding rows of the `air_quality` table.

```
In [19]: air_quality.head()
Out[19]:
                        date.utc            location parameter  value
2067  2019-05-07 01:00:00+00:00  London Westminster       no2   23.0
1003  2019-05-07 01:00:00+00:00             FR04014       no2   25.0
100   2019-05-07 01:00:00+00:00             BETR801      pm25   12.5
1098  2019-05-07 01:00:00+00:00             BETR801       no2   50.5
1109  2019-05-07 01:00:00+00:00  London Westminster      pm25    8.0
```

```
In [20]: air_quality = pd.merge(air_quality, stations_coord,
   ....:                       how='left', on='location')
   ....:

In [21]: air_quality.head()
Out[21]:
                    date.utc            location parameter  value  coordinates.
↪latitude  coordinates.longitude
0  2019-05-07 01:00:00+00:00  London Westminster       no2   23.0              51.
↪49467              -0.13193
1  2019-05-07 01:00:00+00:00             FR04014       no2   25.0              48.
↪83724               2.39390
2  2019-05-07 01:00:00+00:00             FR04014       no2   25.0              48.
↪83722               2.39390
3  2019-05-07 01:00:00+00:00             BETR801      pm25   12.5              51.
↪20966               4.43182
4  2019-05-07 01:00:00+00:00             BETR801       no2   50.5              51.
↪20966               4.43182
```

Using the *`merge()`* function, for each of the rows in the `air_quality` table, the corresponding coordinates are added from the `air_quality_stations_coord` table. Both tables have the column `location` in common which is used as a key to combine the information. By choosing the `left` join, only the locations available in the `air_quality` (left) table, i.e. FR04014, BETR801 and London Westminster, end up in the resulting table. The `merge` function supports multiple join options similar to database-style operations.

Add the parameter full description and name, provided by the parameters metadata table, to the measurements table

Warning: The air quality parameters metadata are stored in a data file `air_quality_parameters.csv`, downloaded using the py-openaq package.

```
In [22]: air_quality_parameters = pd.read_csv("data/air_quality_parameters.csv")

In [23]: air_quality_parameters.head()
Out[23]:
     id                                        description  name
0    bc                                       Black Carbon    BC
1    co                                    Carbon Monoxide    CO
2   no2                                   Nitrogen Dioxide   NO2
3    o3                                              Ozone    O3
4  pm10  Particulate matter less than 10 micrometers in...  PM10
```

```
In [24]: air_quality = pd.merge(air_quality, air_quality_parameters,
   ....:                        how='left', left_on='parameter', right_on='id')
   ....:

In [25]: air_quality.head()
Out[25]:
                    date.utc            location parameter  ...    id                                                      ␣
↪                  description   name
0  2019-05-07 01:00:00+00:00  London Westminster       no2  ...   no2                                                      ␣
↪                  Nitrogen Dioxide    NO2
1  2019-05-07 01:00:00+00:00             FR04014       no2  ...   no2                                                      ␣
↪                  Nitrogen Dioxide    NO2
2  2019-05-07 01:00:00+00:00             FR04014       no2  ...   no2                                                      ␣
↪                  Nitrogen Dioxide    NO2
3  2019-05-07 01:00:00+00:00             BETR801      pm25  ...  pm25  Particulate␣
↪matter less than 2.5 micrometers i...  PM2.5
4  2019-05-07 01:00:00+00:00             BETR801       no2  ...   no2                                                      ␣
↪                  Nitrogen Dioxide    NO2

[5 rows x 9 columns]
```

Compared to the previous example, there is no common column name. However, the `parameter` column in the `air_quality` table and the `id` column in the `air_quality_parameters_name` both provide the measured variable in a common format. The `left_on` and `right_on` arguments are used here (instead of just `on`) to make the link between the two tables.

pandas supports also inner, outer, and right joins. More information on join/merge of tables is provided in the user guide section on *database style merging of tables*. Or have a look at the *comparison with SQL* page.

- Multiple tables can be concatenated both column as row wise using the `concat` function.
- For database-like merging/joining of tables, use the `merge` function.

See the user guide for a full description of the various *facilities to combine data tables*.

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt
```

For this tutorial, air quality data about NO_2 and Particulate matter less than 2.5 micrometers is used, made available by openaq and downloaded using the py-openaq package. The `air_quality_no2_long.csv"` data set provides NO_2 values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [3]: air_quality = pd.read_csv("data/air_quality_no2_long.csv")

In [4]: air_quality = air_quality.rename(columns={"date.utc": "datetime"})
```

(continues on next page)

(continued from previous page)

```
In [5]: air_quality.head()
Out[5]:
    city country                   datetime location parameter  value   unit
0  Paris      FR  2019-06-21 00:00:00+00:00  FR04014       no2   20.0  µg/m³
1  Paris      FR  2019-06-20 23:00:00+00:00  FR04014       no2   21.8  µg/m³
2  Paris      FR  2019-06-20 22:00:00+00:00  FR04014       no2   26.5  µg/m³
3  Paris      FR  2019-06-20 21:00:00+00:00  FR04014       no2   24.9  µg/m³
4  Paris      FR  2019-06-20 20:00:00+00:00  FR04014       no2   21.4  µg/m³
```

```
In [6]: air_quality.city.unique()
Out[6]: array(['Paris', 'Antwerpen', 'London'], dtype=object)
```

How to handle time series data with ease?

Using pandas datetime properties

I want to work with the dates in the column `datetime` as datetime objects instead of plain text

```
In [7]: air_quality["datetime"] = pd.to_datetime(air_quality["datetime"])

In [8]: air_quality["datetime"]
Out[8]:
0      2019-06-21 00:00:00+00:00
1      2019-06-20 23:00:00+00:00
2      2019-06-20 22:00:00+00:00
3      2019-06-20 21:00:00+00:00
4      2019-06-20 20:00:00+00:00
                  ...
2063   2019-05-07 06:00:00+00:00
2064   2019-05-07 04:00:00+00:00
2065   2019-05-07 03:00:00+00:00
2066   2019-05-07 02:00:00+00:00
2067   2019-05-07 01:00:00+00:00
Name: datetime, Length: 2068, dtype: datetime64[ns, UTC]
```

Initially, the values in `datetime` are character strings and do not provide any datetime operations (e.g. extract the year, day of the week,...). By applying the `to_datetime` function, pandas interprets the strings and convert these to datetime (i.e. `datetime64[ns, UTC]`) objects. In pandas we call these datetime objects similar to `datetime.datetime` from the standard library a *`pandas.Timestamp`*.

Note: As many data sets do contain datetime information in one of the columns, pandas input function like *`pandas.read_csv()`* and *`pandas.read_json()`* can do the transformation to dates when reading the data using the `parse_dates` parameter with a list of the columns to read as Timestamp:

```
pd.read_csv("../data/air_quality_no2_long.csv", parse_dates=["datetime"])
```

Why are these *`pandas.Timestamp`* objects useful. Let's illustrate the added value with some example cases.

What is the start and end date of the time series data set working with?

```
In [9]: air_quality["datetime"].min(), air_quality["datetime"].max()
Out[9]:
(Timestamp('2019-05-07 01:00:00+0000', tz='UTC'),
 Timestamp('2019-06-21 00:00:00+0000', tz='UTC'))
```

Using *pandas.Timestamp* for datetimes enable us to calculate with date information and make them comparable. Hence, we can use this to get the length of our time series:

```
In [10]: air_quality["datetime"].max() - air_quality["datetime"].min()
Out[10]: Timedelta('44 days 23:00:00')
```

The result is a *pandas.Timedelta* object, similar to `datetime.timedelta` from the standard Python library and defining a time duration.

The different time concepts supported by pandas are explained in the user guide section on *time related concepts*.

I want to add a new column to the `DataFrame` containing only the month of the measurement

```
In [11]: air_quality["month"] = air_quality["datetime"].dt.month

In [12]: air_quality.head()
Out[12]:
    city country                   datetime location parameter  value   unit  month
0  Paris      FR 2019-06-21 00:00:00+00:00  FR04014       no2   20.0  µg/m³      6
1  Paris      FR 2019-06-20 23:00:00+00:00  FR04014       no2   21.8  µg/m³      6
2  Paris      FR 2019-06-20 22:00:00+00:00  FR04014       no2   26.5  µg/m³      6
3  Paris      FR 2019-06-20 21:00:00+00:00  FR04014       no2   24.9  µg/m³      6
4  Paris      FR 2019-06-20 20:00:00+00:00  FR04014       no2   21.4  µg/m³      6
```

By using `Timestamp` objects for dates, a lot of time-related properties are provided by pandas. For example the `month`, but also `year`, `weekofyear`, `quarter`,… All of these properties are accessible by the `dt` accessor.

An overview of the existing date properties is given in the *time and date components overview table*. More details about the `dt` accessor to return datetime like properties is explained in a dedicated section on the *dt accessor*.

What is the average NO_2 concentration for each day of the week for each of the measurement locations?

```
In [13]: air_quality.groupby(
   ....:     [air_quality["datetime"].dt.weekday, "location"])["value"].mean()
   ....:
Out[13]:
datetime  location
0         BETR801               27.875000
          FR04014               24.856250
          London Westminster    23.969697
1         BETR801               22.214286
          FR04014               30.999359
                                  ...
5         FR04014               25.266154
          London Westminster    24.977612
6         BETR801               21.896552
          FR04014               23.274306
          London Westminster    24.859155
Name: value, Length: 21, dtype: float64
```

Remember the split-apply-combine pattern provided by `groupby` from the *tutorial on statistics calculation*? Here, we want to calculate a given statistic (e.g. mean NO_2) **for each weekday** and **for each measurement location**. To group on weekdays, we use the datetime property `weekday` (with Monday=0 and Sunday=6) of pandas `Timestamp`, which is also accessible by the `dt` accessor. The grouping on both locations and weekdays can be done to split the calculation of the mean on each of these combinations.

> **Danger:** As we are working with a very short time series in these examples, the analysis does not provide a long-term representative result!

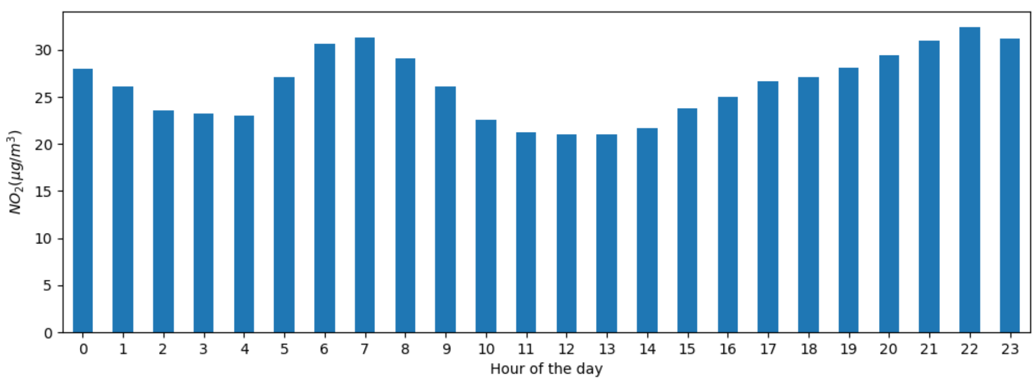
Plot the typical NO_2 pattern during the day of our time series of all stations together. In other words, what is the average value for each hour of the day?

```
In [14]: fig, axs = plt.subplots(figsize=(12, 4))

In [15]: air_quality.groupby(
   ....:     air_quality["datetime"].dt.hour)["value"].mean().plot(kind='bar',
   ....:                                                          rot=0,
   ....:                                                          ax=axs)
   ....:
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5344257f90>

In [16]: plt.xlabel("Hour of the day");  # custom x label using matplotlib

In [17]: plt.ylabel("$NO_2 (µg/m^3)$");
```



Similar to the previous case, we want to calculate a given statistic (e.g. mean NO_2) **for each hour of the day** and we can use the split-apply-combine approach again. For this case, the datetime property `hour` of pandas `Timestamp`, which is also accessible by the `dt` accessor.

Datetime as index

In the *tutorial on reshaping*, `pivot()` was introduced to reshape the data table with each of the measurements locations as a separate column:

```
In [18]: no_2 = air_quality.pivot(index="datetime", columns="location", values="value
↪")

In [19]: no_2.head()
Out[19]:
location                   BETR801  FR04014  London Westminster
datetime
2019-05-07 01:00:00+00:00     50.5     25.0                23.0
```

(continues on next page)

(continued from previous page)

```
2019-05-07 02:00:00+00:00     45.0     27.7               19.0
2019-05-07 03:00:00+00:00      NaN     50.4               19.0
2019-05-07 04:00:00+00:00      NaN     61.9               16.0
2019-05-07 05:00:00+00:00      NaN     72.4                NaN
```

Note: By pivoting the data, the datetime information became the index of the table. In general, setting a column as an index can be achieved by the `set_index` function.

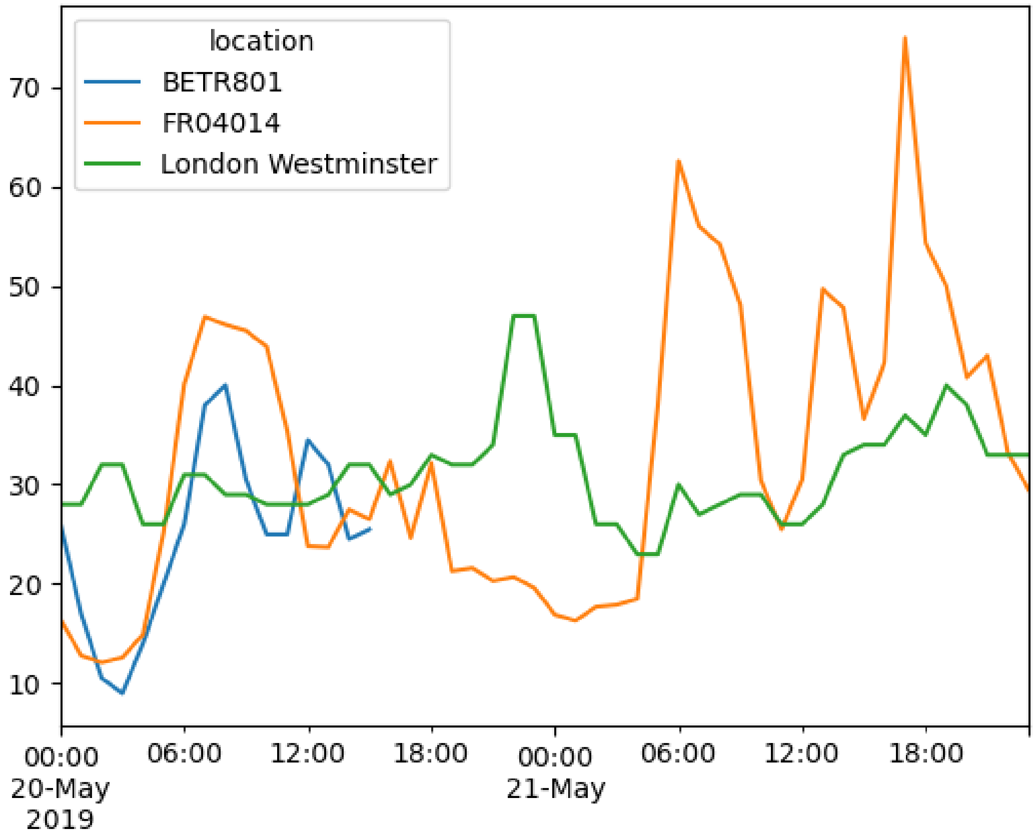
Working with a datetime index (i.e. `DatetimeIndex`) provides powerful functionalities. For example, we do not need the `dt` accessor to get the time series properties, but have these properties available on the index directly:

```
In [20]: no_2.index.year, no_2.index.weekday
Out[20]:
(Int64Index([2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019,
             ...
             2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019],
            dtype='int64', name='datetime', length=1033),
 Int64Index([1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             ...
             3, 3, 3, 3, 3, 3, 3, 3, 3, 4],
            dtype='int64', name='datetime', length=1033))
```

Some other advantages are the convenient subsetting of time period or the adapted time scale on plots. Let's apply this on our data.

Create a plot of the NO_2 values in the different stations from the 20th of May till the end of 21st of May

```
In [21]: no_2["2019-05-20":"2019-05-21"].plot();
```



By providing a **string that parses to a datetime**, a specific subset of the data can be selected on a `DatetimeIndex`.

More information on the `DatetimeIndex` and the slicing by using strings is provided in the section on *time series indexing*.

Resample a time series to another frequency

Aggregate the current hourly time series values to the monthly maximum value in each of the stations.

```
In [22]: monthly_max = no_2.resample("M").max()

In [23]: monthly_max
Out[23]:
location                   BETR801  FR04014  London Westminster
datetime
2019-05-31 00:00:00+00:00     74.5     97.0                97.0
2019-06-30 00:00:00+00:00     52.5     84.7                52.0
```

A very powerful method on time series data with a datetime index, is the ability to `resample()` time series to another frequency (e.g., converting secondly data into 5-minutely data).

The `resample()` method is similar to a groupby operation:

- it provides a time-based grouping, by using a string (e.g. `M`, `5H`,. . .) that defines the target frequency
- it requires an aggregation function such as `mean`, `max`,. . .

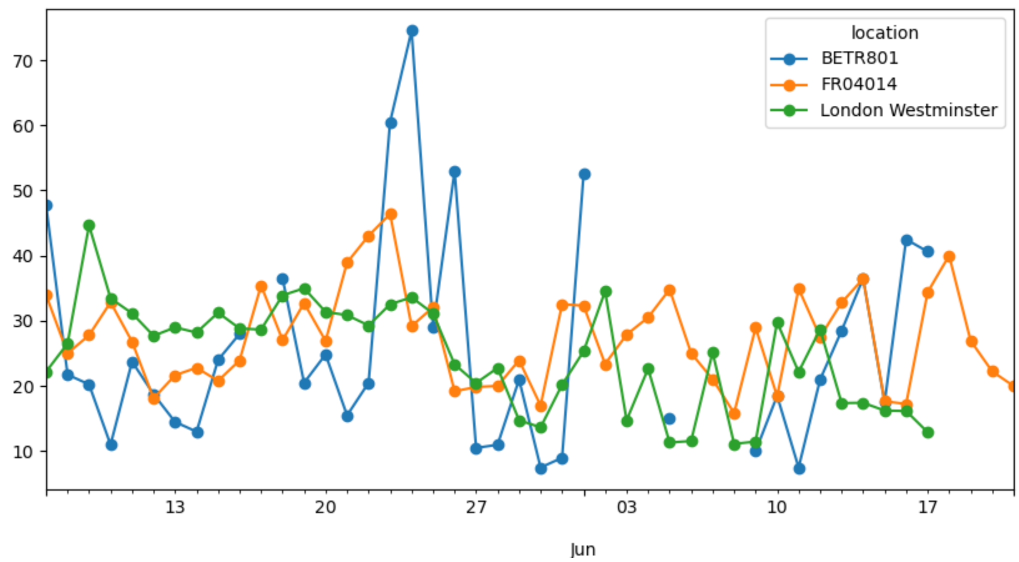
An overview of the aliases used to define time series frequencies is given in the *offset aliases overview table*.

When defined, the frequency of the time series is provided by the `freq` attribute:

```
In [24]: monthly_max.index.freq
Out[24]: <MonthEnd>
```

Make a plot of the daily median NO_2 value in each of the stations.

```
In [25]: no_2.resample("D").mean().plot(style="-o", figsize=(10, 5));
```



More details on the power of time series `resampling` is provided in the user gudie section on *resampling*.

- Valid date strings can be converted to datetime objects using `to_datetime` function or as part of read functions.
- Datetime objects in pandas supports calculations, logical operations and convenient date-related properties using the `dt` accessor.
- A `DatetimeIndex` contains these date-related properties and supports convenient slicing.
- `Resample` is a powerful method to change the frequency of a time series.

A full overview on time series is given in the pages on *time series and date functionality*.

```
In [1]: import pandas as pd
```

This tutorial uses the titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.

- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass                                               Name  ␣
↪  Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris  ␣
↪ male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina  ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry  ␣
↪ male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

How to manipulate textual data?

Make all name characters lowercase

```
In [4]: titanic["Name"].str.lower()
Out[4]:
0                                braund, mr. owen harris
1      cumings, mrs. john bradley (florence briggs th...
2                                 heikkinen, miss. laina
3           futrelle, mrs. jacques heath (lily may peel)
4                               allen, mr. william henry
                             ...
886                                montvila, rev. juozas
887                         graham, miss. margaret edith
888             johnston, miss. catherine helen "carrie"
889                                behr, mr. karl howell
890                                  dooley, mr. patrick
Name: Name, Length: 891, dtype: object
```

To make each of the strings in the `Name` column lowercase, select the `Name` column (see *tutorial on selection of data*), add the `str` accessor and apply the `lower` method. As such, each of the strings is converted element wise.

Similar to datetime objects in the *time series tutorial* having a `dt` accessor, a number of specialized string methods are available when using the `str` accessor. These methods have in general matching names with the equivalent built-in string methods for single elements, but are applied element-wise (remember *element wise calculations*?) on each of the values of the columns.

Create a new column `Surname` that contains the surname of the Passengers by extracting the part before the comma.

```
In [5]: titanic["Name"].str.split(",")
Out[5]:
0                             [Braund,  Mr. Owen Harris]
1      [Cumings,  Mrs. John Bradley (Florence Briggs ...
2                              [Heikkinen,  Miss. Laina]
3        [Futrelle,  Mrs. Jacques Heath (Lily May Peel)]
4                            [Allen,  Mr. William Henry]
                             ...
886                             [Montvila,  Rev. Juozas]
887                      [Graham,  Miss. Margaret Edith]
888          [Johnston,  Miss. Catherine Helen "Carrie"]
889                             [Behr,  Mr. Karl Howell]
890                               [Dooley,  Mr. Patrick]
Name: Name, Length: 891, dtype: object
```

Using the *Series.str.split()* method, each of the values is returned as a list of 2 elements. The first element is the part before the comma and the second element the part after the comma.

```
In [6]: titanic["Surname"] = titanic["Name"].str.split(",").str.get(0)

In [7]: titanic["Surname"]
Out[7]:
0         Braund
1        Cumings
2      Heikkinen
3       Futrelle
4          Allen
         ...
886     Montvila
887       Graham
888     Johnston
889         Behr
890       Dooley
Name: Surname, Length: 891, dtype: object
```

As we are only interested in the first part representing the surname (element 0), we can again use the `str` accessor and apply *Series.str.get()* to extract the relevant part. Indeed, these string functions can be concatenated to combine multiple functions at once!

More information on extracting parts of strings is available in the user guide section on *splitting and replacing strings*.

Extract the passenger data about the Countess on board of the Titanic.

```
In [8]: titanic["Name"].str.contains("Countess")
Out[8]:
0      False
1      False
2      False
3      False
4      False
       ...
886    False
887    False
888    False
889    False
890    False
Name: Name, Length: 891, dtype: bool
```

```
In [9]: titanic[titanic["Name"].str.contains("Countess")]
Out[9]:
     PassengerId  Survived  Pclass                                               Name␣
↪     Sex  ...  Ticket  Fare  Cabin Embarked  Surname
759          760         1       1  Rothes, the Countess. of (Lucy Noel Martha Dye...␣
↪ female  ...  110152  86.5    B77        S   Rothes

[1 rows x 13 columns]
```

(*Interested in her story? See*Wikipedia*!*)

The string method *Series.str.contains()* checks for each of the values in the column `Name` if the string contains the word `Countess` and returns for each of the values `True` (`Countess` is part of the name) of `False` (`Countess` is notpart of the name). This output can be used to subselect the data using conditional (boolean) indexing introduced in the *subsetting of data tutorial*. As there was only 1 Countess on the Titanic, we get one row as a result.

Note: More powerful extractions on strings is supported, as the *Series.str.contains()* and *Series.str.extract()* methods accepts regular expressions, but out of scope of this tutorial.

More information on extracting parts of strings is available in the user guide section on *string matching and extracting*.

Which passenger of the titanic has the longest name?

```
In [10]: titanic["Name"].str.len()
Out[10]:
0      23
1      51
2      22
3      44
4      24
       ..
886    21
887    28
888    40
889    21
890    19
Name: Name, Length: 891, dtype: int64
```

To get the longest name we first have to get the lenghts of each of the names in the `Name` column. By using pandas string methods, the *Series.str.len()* function is applied to each of the names individually (element-wise).

```
In [11]: titanic["Name"].str.len().idxmax()
Out[11]: 307
```

Next, we need to get the corresponding location, preferably the index label, in the table for which the name length is the largest. The `idxmax`()` method does exactly that. It is not a string method and is applied to integers, so no `str` is used.

```
In [12]: titanic.loc[titanic["Name"].str.len().idxmax(), "Name"]
Out[12]: 'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y␣
↪Vallejo)'
```

Based on the index name of the row (`307`) and the column (`Name`), we can do a selection using the `loc` operator, introduced in the tutorial on subsetting.

In the ‘Sex’ columns, replace values of ‘male’ by ‘M’ and all ‘female’ values by ‘F’

```
In [13]: titanic["Sex_short"] = titanic["Sex"].replace({"male": "M",
   ....:                                                 "female": "F"})
   ....:

In [14]: titanic["Sex_short"]
Out[14]:
0      M
1      F
2      F
3      F
4      M
      ..
886    M
887    F
888    F
889    M
890    M
Name: Sex_short, Length: 891, dtype: object
```

Whereas *replace()* is not a string method, it provides a convenient way to use mappings or vocabularies to translate certain values. It requires a `dictionary` to define the mapping `{from : to}`.

Warning: There is also a *replace()* methods available to replace a specific set of characters. However, when having a mapping of multiple values, this would become:

```
titanic["Sex_short"] = titanic["Sex"].str.replace("female", "F")
titanic["Sex_short"] = titanic["Sex_short"].str.replace("male", "M")
```

This would become cumbersome and easily lead to mistakes. Just think (or try out yourself) what would happen if those two statements are applied in the opposite order. . .

- String methods are available using the `str` accessor.
- String methods work element wise and can be used for conditional indexing.
- The `replace` method is a convenient method to convert values according to a given dictionary.

A full overview is provided in the user guide pages on *working with text data*.

1.4.5 Essential basic functionality

Here we discuss a lot of the essential functionality common to the pandas data structures. Here's how to create some of the objects used in the examples from the previous section:

```
In [1]: index = pd.date_range('1/1/2000', periods=8)

In [2]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   ...:                   columns=['A', 'B', 'C'])
   ...:
```

Head and tail

To view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```
In [4]: long_series = pd.Series(np.random.randn(1000))

In [5]: long_series.head()
Out[5]:
0   -1.157892
1   -1.344312
2    0.844885
3    1.075770
4   -0.109050
dtype: float64

In [6]: long_series.tail(3)
Out[6]:
997   -0.289388
998   -1.020544
999    0.589993
dtype: float64
```

Attributes and underlying data

pandas objects have a number of attributes enabling you to access the metadata

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- **Axis labels**
  - **Series**: *index* (only axis)
  - **DataFrame**: *index* (rows) and *columns*

Note, **these attributes can be safely assigned to**!

```
In [7]: df[:2]
Out[7]:
                   A         B         C
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929

In [8]: df.columns = [x.lower() for x in df.columns]

In [9]: df
Out[9]:
                   a         b         c
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929
2000-01-03  1.071804  0.721555 -0.706771
2000-01-04 -1.039575  0.271860 -0.424972
2000-01-05  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427
2000-01-07  0.524988  0.404705  0.577046
2000-01-08 -1.715002 -1.039268 -0.370647
```

Pandas objects (`Index`, `Series`, `DataFrame`) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a `numpy.ndarray`. However, pandas and 3rd party libraries may *extend* NumPy's type system to add support for custom arrays (see *dtypes*).

To get the actual data inside a `Index` or `Series`, use the `.array` property

```
In [10]: s.array
Out[10]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
 -1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64

In [11]: s.index.array
Out[11]:
<PandasArray>
['a', 'b', 'c', 'd', 'e']
Length: 5, dtype: object
```

`array` will always be an `ExtensionArray`. The exact details of what an `ExtensionArray` is and why pandas uses them is a bit beyond the scope of this introduction. See *dtypes* for more.

If you know you need a NumPy array, use `to_numpy()` or `numpy.asarray()`.

```
In [12]: s.to_numpy()
Out[12]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])

In [13]: np.asarray(s)
Out[13]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

When the Series or Index is backed by an `ExtensionArray`, `to_numpy()` may involve copying data and coercing values. See *dtypes* for more.

`to_numpy()` gives some control over the `dtype` of the resulting `numpy.ndarray`. For example, consider datetimes with timezones. NumPy doesn't have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations:

1. An object-dtype `numpy.ndarray` with `Timestamp` objects, each with the correct `tz`
2. A `datetime64[ns]` -dtype `numpy.ndarray`, where the values have been converted to UTC and the timezone discarded

Timezones may be preserved with `dtype=object`

```
In [14]: ser = pd.Series(pd.date_range('2000', periods=2, tz="CET"))

In [15]: ser.to_numpy(dtype=object)
Out[15]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET', freq='D'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET', freq='D')],
      dtype=object)
```

Or thrown away with `dtype='datetime64[ns]'`

```
In [16]: ser.to_numpy(dtype="datetime64[ns]")
Out[16]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
      dtype='datetime64[ns]')
```

Getting the "raw data" inside a `DataFrame` is possibly a bit more complex. When your `DataFrame` only has a single data type for all the columns, `DataFrame.to_numpy()` will return the underlying data:

```
In [17]: df.to_numpy()
Out[17]:
array([[-0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949],
       [ 1.0718,  0.7216, -0.7068],
       [-1.0396,  0.2719, -0.425 ],
       [ 0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784],
       [ 0.525 ,  0.4047,  0.577 ],
       [-1.715 , -1.0393, -0.3706]])
```

If a DataFrame contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame's columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

Note: When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

In the past, pandas recommended `Series.values` or `DataFrame.values` for extracting the data from a Series or DataFrame. You'll still find references to these in old code bases and online. Going forward, we recommend avoiding `.values` and using `.array` or `.to_numpy()`. `.values` has the following drawbacks:

1. When your Series contains an *extension type*, it's unclear whether `Series.values` returns a NumPy array or the extension array. `Series.array` will always return an `ExtensionArray`, and will never copy data. `Series.to_numpy()` will always return a NumPy array, potentially at the cost of copying / coercing values.
2. When your DataFrame contains a mixture of data types, `DataFrame.values` may involve copying data and coercing values to a common dtype, a relatively expensive operation. `DataFrame.to_numpy()`, being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame.

Accelerated operations

pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have `nans`.

Here is a sample (using 100 column x 100,000 row `DataFrames`):

Operation	0.11.0 (ms)	Prior Version (ms)	Ratio to Prior
`df1 > df2`	13.32	125.35	0.1063
`df1 * df2`	21.71	36.63	0.5928
`df1 + df2`	22.04	36.50	0.6039

You are highly encouraged to install both libraries. See the section *Recommended Dependencies* for more installation info.

These are both enabled to be used by default, you can control this by setting the options:

```
pd.set_option('compute.use_bottleneck', False)
pd.set_option('compute.use_numexpr', False)
```

Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects.
- Missing data in computations.

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, … for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the **axis** keyword:

```
In [18]: df = pd.DataFrame({
   ....:     'one': pd.Series(np.random.randn(3), index=['a', 'b', 'c']),
   ....:     'two': pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   ....:     'three': pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   ....:

In [19]: df
Out[19]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [20]: row = df.iloc[1]

In [21]: column = df['two']

In [22]: df.sub(row, axis='columns')
Out[22]:
        one       two     three
a  1.051928 -0.139606       NaN
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d       NaN -1.632779 -0.562782

In [23]: df.sub(row, axis=1)
Out[23]:
        one       two     three
a  1.051928 -0.139606       NaN
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d       NaN -1.632779 -0.562782

In [24]: df.sub(column, axis='index')
Out[24]:
```

(continues on next page)

(continued from previous page)

```
        one  two     three
a -0.377535  0.0       NaN
b -1.569069  0.0 -1.962513
c -0.783123  0.0 -0.250933
d       NaN  0.0 -0.892516

In [25]: df.sub(column, axis=0)
Out[25]:
        one  two     three
a -0.377535  0.0       NaN
b -1.569069  0.0 -1.962513
c -0.783123  0.0 -0.250933
d       NaN  0.0 -0.892516
```

Furthermore you can align a level of a MultiIndexed DataFrame with a Series.

```
In [26]: dfmi = df.copy()

In [27]: dfmi.index = pd.MultiIndex.from_tuples([(1, 'a'), (1, 'b'),
   ....:                                          (1, 'c'), (2, 'a')],
   ....:                                         names=['first', 'second'])
   ....:

In [28]: dfmi.sub(column, axis=0, level='second')
Out[28]:
                   one       two     three
first second
1     a      -0.377535  0.000000       NaN
      b      -1.569069  0.000000 -1.962513
      c      -0.783123  0.000000 -0.250933
2     a            NaN -1.493173 -2.385688
```

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

```
In [29]: s = pd.Series(np.arange(10))

In [30]: s
Out[30]:
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64

In [31]: div, rem = divmod(s, 3)

In [32]: div
Out[32]:
0    0
```

(continues on next page)