05_pandas

November 15, 2023

1 Pandas

The Numpy library is excellent for numerical computations, but it lacks support to handle missing data or non-omogeneous arrays. The **Pandas** library is based on Numpy and extends the Numpy functionality, and is currently one of the most widely used tools for data manipulation, providing high-performance, easy-to-use data structures and advanced data analysis tools.

In particular Pandas features:

- A fast and efficient `Series` and `DataFrame` objects for data manipulation with integrated indexing;
- Tools for reading and writing data between in-memory data structures and different formats (CSV, Excel, SQL, HDF5);
- Smart data alignment and integrated handling of missing data;
- Time series-functionalities;
- Convenient label-based slicing, fancy indexing, and subsetting of large data sets;
- Hierarchical axis indexing provides an intuitive way of working with high-dimensional data in a lower-dimensional data structure;
- Aggregating and transforming data with a powerful "group-by" engine;
- High performance merging and joining of data sets;
- Highly optimized for performance, with critical code paths written in Cython or C.

[1]:
```
import pandas as pd # standard naming convention
import numpy as np
```

1.1 Series

Pandas Series represent an extension of the Numpy 1D arrays. The content of a Series is equivalent to a Numpy array, and in addition the axis is labeled. Labels don't need to be unique, but must be of a *hashable* type.

Since the content is of type `ndarray`, the content has to be *omogeneous*. However, there is the possibility to store heterogeneous data, but the content in this case would be of type `object`.

One of the most important examples are the **time series**, which are used to keep track of the time evolution of a certain quantity.

Link to the official Pandas Series documentation.

[2]:
```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

# Calling the Series constructor
# Constructor requires the data, and optionally the indices and data type
sr = pd.Series(np.arange(10)*0.5, index=tuple(letters[:10]), dtype=float)
print("series:\n", sr, '\n')
print("series type:\n", type(sr), '\n')
print("indices:\n", sr.index, '\n')
print("values:", sr.values, type(sr.values), '\n') # values of the Series are␣
 ↪actually a numpy array
print("type:\n", sr.dtype, '\n')
```

```
series:
 a    0.0
b    0.5
c    1.0
d    1.5
e    2.0
f    2.5
g    3.0
h    3.5
i    4.0
j    4.5
dtype: float64

series type:
 <class 'pandas.core.series.Series'>

indices:
 Index(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype='object')

values: [0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5] <class 'numpy.ndarray'>

type:
 float64
```

[3]:
```
print("element by index:", sr['f'], '\n') # Accessing elements like arrays
print("element by attribute:", sr.f, '\n') # Accessing elements like attributes␣
 ↪- not recommended

# selecting a subset of the series
subsr = sr[['d', 'f', 'h']] # note the double square brackets
print("series subset:\n", subsr, type(subsr), '\n') # Multiple indexing returns␣
 ↪another series
```

```
element by index: 2.5
```

```
element by attribute: 2.5

series subset:
 d    1.5
f    2.5
h    3.5
dtype: float64 <class 'pandas.core.series.Series'>
```

[4]:
```
# Extracting elements and operations are the same as numpy array (slicing)
print(sr[:3], '\n')
print(sr[7:], '\n')
print(sr[::3], '\n')

# Fancy indexing works on Series, too
print(sr[sr > 3], '\n')

# You can also pass Series to numpy funtions
print(np.exp(sr), '\n')
print("Series mean:", np.mean(sr), ", std:", np.std(sr), '\n')
```

```
a    0.0
b    0.5
c    1.0
dtype: float64

h    3.5
i    4.0
j    4.5
dtype: float64

a    0.0
d    1.5
g    3.0
j    4.5
dtype: float64

h    3.5
i    4.0
j    4.5
dtype: float64

a     1.000000
b     1.648721
c     2.718282
d     4.481689
e     7.389056
```

```
f    12.182494
g    20.085537
h    33.115452
i    54.598150
j    90.017131
dtype: float64

Series mean: 2.25 , std: 1.4361406616345072
```

Series may contain non-omogeneous data; in this case, the data type is referred to as `object`. Non-homogeneous data is normally handeled also by Pandas and does not represent a problem, however this pays the price of less time-efficient operations.

[5]:
```
# Series can be created from a python dictionary, too
# Note that the elements can be of different types
d = {'b' : 1, 'a' : 'cat', 'c' : [2, 3]}
so = pd.Series(d) # alternative constructor that taks a dict as the only input
print(so, '\n')
```

```
b         1
a       cat
c    [2, 3]
dtype: object
```

A key difference between Pandas Series and Numpy arrays is that operations between Series **automatically align the data based on the label**.

Thus, you can write operations without considering whether the Series involved have the same labels, or even the same size.

If there is no matching element, the resulting value would be a `NaN`.

[6]:
```
s = pd.Series(np.arange(5), index=tuple(letters[:5]))
print("series:\n", s, '\n')

s1 = s[1:]
print("shifted series:\n", s1, '\n')

s2 = s1 + s
print("shifted sum:\n", s2, '\n')

s3 = s1 + s[:-1]
print("double shifted sum:\n", s3, '\n')
```

```
series:
 a    0
b    1
c    2
```

```
d    3
e    4
dtype: int64

shifted series:
 b    1
c    2
d    3
e    4
dtype: int64

shifted sum:
 a    NaN
b    2.0
c    4.0
d    6.0
e    8.0
dtype: float64

double shifted sum:
 a    NaN
b    2.0
c    4.0
d    6.0
e    NaN
dtype: float64
```

1.1.1 Time series

Datetime

When dealing with time, Python provides the `datetime` library that allows to store the date and time in an dedicated object, which possess several methods to access the relevant quantities (day, month, year, hours, minutes, seconds, …)

[7]:
```python
# To define a date, the datetime module is very useful
import datetime as dt

date = dt.date.today()
print("Today's date:", date)

# specify year, month, day, hour, minutes, seconds, and microseconds
date = dt.datetime(2020, 11, 12, 10, 45, 10, 15)
print("Date and time:", date)
print("Month:", date.month, "and minutes:", date.minute)
```

```
Today's date: 2023-11-15
Date and time: 2020-11-12 10:45:10.000015
```

```
Month: 11 and minutes: 45
```

Pandas Timestamps

Timestamped data is the most basic type of time series data that associates values with points in time.

Functions like `pd.to_datetime` can be used to convert between different formats and, for instance, when reading the time stored as a string from a dataset:

[8]:
```
# Get the timestamp, which is the nanoseconds from January 1st 1970 (Unix time)
tstamp = pd.Timestamp(date)
#tstamp = pd.Timestamp(dt.datetime(1970, 1, 1, 0, 0, 0, 1))
print("Timestamp:", tstamp.value)

# when creating a timestamp the format can be explicitly passed
ts = pd.to_datetime('2010/11/12', format='%Y/%m/%d')
print("Time:", ts, ", timestamp:", ts.value, ", type:", type(ts))

ts = pd.to_datetime('12-11-2010 10:39', format='%d-%m-%Y %H:%M')
print("Time:", ts, ", timestamp:", ts.value, ", type:", type(ts))
```

```
Timestamp: 1605177910000015000
Time: 2010-11-12 00:00:00 , timestamp: 1289520000000000000 , type: <class
'pandas._libs.tslibs.timestamps.Timestamp'>
Time: 2010-11-12 10:39:00 , timestamp: 1289558340000000000 , type: <class
'pandas._libs.tslibs.timestamps.Timestamp'>
```

Pandas Date range

Time series are very often used to describe the behaviour of a quantity as a function of time. Pandas has a special type of index for that, `DatetimeIndex`, that can be created e.g. with the function `pd.data_range()`.

[9]:
```
# create DatetimeIndex using ranges:
days = pd.date_range(date, periods=7, freq='D')
print("7 days range:", days)

seconds = pd.date_range(date, periods=3600, freq='s')
print("1 hour in seconds:", seconds)
```

```
7 days range: DatetimeIndex(['2020-11-12 10:45:10.000015', '2020-11-13
10:45:10.000015',
               '2020-11-14 10:45:10.000015', '2020-11-15 10:45:10.000015',
               '2020-11-16 10:45:10.000015', '2020-11-17 10:45:10.000015',
               '2020-11-18 10:45:10.000015'],
              dtype='datetime64[ns]', freq='D')
1 hour in seconds: DatetimeIndex(['2020-11-12 10:45:10.000015', '2020-11-12
10:45:11.000015',
               '2020-11-12 10:45:12.000015', '2020-11-12 10:45:13.000015',
               '2020-11-12 10:45:14.000015', '2020-11-12 10:45:15.000015',
```

```
               '2020-11-12 10:45:16.000015', '2020-11-12 10:45:17.000015',
               '2020-11-12 10:45:18.000015', '2020-11-12 10:45:19.000015',
               …
               '2020-11-12 11:45:00.000015', '2020-11-12 11:45:01.000015',
               '2020-11-12 11:45:02.000015', '2020-11-12 11:45:03.000015',
               '2020-11-12 11:45:04.000015', '2020-11-12 11:45:05.000015',
               '2020-11-12 11:45:06.000015', '2020-11-12 11:45:07.000015',
               '2020-11-12 11:45:08.000015', '2020-11-12 11:45:09.000015'],
              dtype='datetime64[ns]', length=3600, freq='S')
```

To learn more about the frequency strings, please check the documentation.

A standard series can be created, and (a range of) elements can be used as indices:

[10]:
```
print("index:\n", days, '\n')
tseries = pd.Series(np.random.normal(10, 1, len(days)), index=days)
print("time series:\n", days, '\n')
# Extracting elements
print("slice by position:\n", tseries[0:4], '\n')
print("slice by date range:\n", tseries['2020-9-11' : '2020-9-14'], '\n') #␣
 ↪note that includes end time
```

```
index:
 DatetimeIndex(['2020-11-12 10:45:10.000015', '2020-11-13 10:45:10.000015',
               '2020-11-14 10:45:10.000015', '2020-11-15 10:45:10.000015',
               '2020-11-16 10:45:10.000015', '2020-11-17 10:45:10.000015',
               '2020-11-18 10:45:10.000015'],
              dtype='datetime64[ns]', freq='D')

time series:
 DatetimeIndex(['2020-11-12 10:45:10.000015', '2020-11-13 10:45:10.000015',
               '2020-11-14 10:45:10.000015', '2020-11-15 10:45:10.000015',
               '2020-11-16 10:45:10.000015', '2020-11-17 10:45:10.000015',
               '2020-11-18 10:45:10.000015'],
              dtype='datetime64[ns]', freq='D')

slice by position:
 2020-11-12 10:45:10.000015     9.893543
2020-11-13 10:45:10.000015     9.886764
2020-11-14 10:45:10.000015     9.128595
2020-11-15 10:45:10.000015    11.150927
Freq: D, dtype: float64

slice by date range:
 Series([], Freq: D, dtype: float64)
```

`pd.to_datetime` can also be used to create a `DatetimeIndex` if the argument is a list:

[11]:
```
print(pd.to_datetime([1, 2, 3, 4], unit='D', origin=pd.Timestamp('1980-02-03')))
```

```
DatetimeIndex(['1980-02-04', '1980-02-05', '1980-02-06', '1980-02-07'],
dtype='datetime64[ns]', freq=None)
```

1.2 DataFrame

A pandas DataFrame can be thought as a tabular spreadsheet, although the performance, the functionalities and the capabilities are very different.

Similarly to Series, the DataFrame structure also contains labeled axes (rows and columns). Arithmetic operations **align on both row and column labels**. Each column in a DataFrame is a Series object: as a matter of fact, a DataFrame can be thought of as a dict-like container for Series objects.

The elements can be of all types, and missing data could be present too (represented as `NaN`).

For future reference (or for people already familiar with R), a pandas DataFrame is also similar to the R DataFrame.

Link to the official documentation.

1.2.1 Constructor

A DataFrame objects can be created by passing a dictionary of objects. Note that the dictionary values are not omogeneous and do not have the same length. In these cases, pandas will automatically adjust the sizes, by replicating the content or adding NaN if necessary.

[12]:
```
df = pd.DataFrame({
    'A' : 1.,
    'B' : pd.Timestamp('20130102'),
    'C' : pd.Series(3, index=range(4), dtype='float32'),
    'D' : np.arange(7, 11),
    'E' : pd.Categorical(["test", "train", "test", "train"]), # a Series that␣
 ↪represents a category label
})
# the keys of the dictionary represent the labels of the columns
# since no index is specified, the simplest one [0, 1, 2, ...] is added by␣
 ↪Pandas automatically

df
```

[12]:

	A	B	C	D	E
0	1.0	2013-01-02	3.0	7	test
1	1.0	2013-01-02	3.0	8	train
2	1.0	2013-01-02	3.0	9	test
3	1.0	2013-01-02	3.0	10	train

An example of DataFrame with a `DatatimeIndex` object as index:

[13]:
```
entries = 10
columns = ['A', 'B', 'C', 'D']
dates = pd.date_range('11/9/2020 14:45:00', freq='h', periods=entries) # days/
 ↪month/year
df = pd.DataFrame(np.random.randn(entries, len(columns)), index=dates,␣
 ↪columns=columns)
df # pay attention that the date is printed as year-day-month
```

[13]:

	A	B	C	D
2020-11-09 14:45:00	0.379644	2.386360	-0.197379	-0.167745
2020-11-09 15:45:00	-0.375580	-0.277857	-0.267869	-0.073870
2020-11-09 16:45:00	0.326133	-0.454830	-0.353803	-0.181075
2020-11-09 17:45:00	-0.869288	-0.664001	-2.575753	1.500908
2020-11-09 18:45:00	-0.459613	1.252780	-0.921579	0.234116
2020-11-09 19:45:00	-0.826298	-1.344278	-0.234228	-1.581432
2020-11-09 20:45:00	0.809296	1.443988	1.200433	-1.364293
2020-11-09 21:45:00	1.624421	1.618971	1.290937	-1.949003
2020-11-09 22:45:00	1.621564	0.524994	1.669798	-0.112020
2020-11-09 23:45:00	-0.400006	-0.000111	-0.910490	1.020831

1.2.2 Viewing Data

[14]:
```
df.head()
```

[14]:

	A	B	C	D
2020-11-09 14:45:00	0.379644	2.386360	-0.197379	-0.167745
2020-11-09 15:45:00	-0.375580	-0.277857	-0.267869	-0.073870
2020-11-09 16:45:00	0.326133	-0.454830	-0.353803	-0.181075
2020-11-09 17:45:00	-0.869288	-0.664001	-2.575753	1.500908
2020-11-09 18:45:00	-0.459613	1.252780	-0.921579	0.234116

[15]:
```
df.tail(4)
```

[15]:

	A	B	C	D
2020-11-09 20:45:00	0.809296	1.443988	1.200433	-1.364293
2020-11-09 21:45:00	1.624421	1.618971	1.290937	-1.949003
2020-11-09 22:45:00	1.621564	0.524994	1.669798	-0.112020
2020-11-09 23:45:00	-0.400006	-0.000111	-0.910490	1.020831

[16]:
```
df.index
```

[16]:
```
DatetimeIndex(['2020-11-09 14:45:00', '2020-11-09 15:45:00',
               '2020-11-09 16:45:00', '2020-11-09 17:45:00',
               '2020-11-09 18:45:00', '2020-11-09 19:45:00',
               '2020-11-09 20:45:00', '2020-11-09 21:45:00',
               '2020-11-09 22:45:00', '2020-11-09 23:45:00'],
              dtype='datetime64[ns]', freq='H')
```

[17]:
```
df.columns
```

[17]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

[18]:
```
df.values
```

[18]:
```
array([[ 3.79644182e-01,  2.38636046e+00, -1.97378889e-01,
        -1.67745138e-01],
       [-3.75580317e-01, -2.77857251e-01, -2.67869287e-01,
        -7.38698437e-02],
       [ 3.26132749e-01, -4.54830454e-01, -3.53803425e-01,
        -1.81075129e-01],
       [-8.69288459e-01, -6.64000723e-01, -2.57575326e+00,
         1.50090781e+00],
       [-4.59613334e-01,  1.25277980e+00, -9.21579306e-01,
         2.34116397e-01],
       [-8.26298180e-01, -1.34427766e+00, -2.34228041e-01,
        -1.58143165e+00],
       [ 8.09295959e-01,  1.44398821e+00,  1.20043270e+00,
        -1.36429349e+00],
       [ 1.62442106e+00,  1.61897057e+00,  1.29093737e+00,
        -1.94900285e+00],
       [ 1.62156391e+00,  5.24993921e-01,  1.66979791e+00,
        -1.12019536e-01],
       [-4.00006461e-01, -1.11465023e-04, -9.10489576e-01,
         1.02083144e+00]])
```

[19]:
```
df.describe()
```

[19]:

	A	B	C	D
count	10.000000	10.000000	10.000000	10.000000
mean	0.183027	0.448602	-0.129993	-0.267358
std	0.930318	1.190965	1.261849	1.099222
min	-0.869288	-1.344278	-2.575753	-1.949003
25%	-0.444712	-0.410587	-0.771318	-1.068489
50%	-0.024724	0.262441	-0.251049	-0.139882
75%	0.701883	1.396186	0.850980	0.157120
max	1.624421	2.386360	1.669798	1.500908

[20]:
```
df.T
```

[20]:

	2020-11-09 14:45:00	2020-11-09 15:45:00	2020-11-09 16:45:00	\
A	0.379644	-0.375580	0.326133	
B	2.386360	-0.277857	-0.454830	
C	-0.197379	-0.267869	-0.353803	
D	-0.167745	-0.073870	-0.181075	

```
   2020-11-09 17:45:00  2020-11-09 18:45:00  2020-11-09 19:45:00  \
A            -0.869288            -0.459613            -0.826298
B            -0.664001             1.252780            -1.344278
C            -2.575753            -0.921579            -0.234228
D             1.500908             0.234116            -1.581432

   2020-11-09 20:45:00  2020-11-09 21:45:00  2020-11-09 22:45:00  \
A             0.809296             1.624421             1.621564
B             1.443988             1.618971             0.524994
C             1.200433             1.290937             1.669798
D            -1.364293            -1.949003            -0.112020

   2020-11-09 23:45:00
A            -0.400006
B            -0.000111
C            -0.910490
D             1.020831
```

[21]:
```
df.sort_index(axis=1, ascending=False)
```

[21]:
```
                            D         C         B         A
2020-11-09 14:45:00 -0.167745 -0.197379  2.386360  0.379644
2020-11-09 15:45:00 -0.073870 -0.267869 -0.277857 -0.375580
2020-11-09 16:45:00 -0.181075 -0.353803 -0.454830  0.326133
2020-11-09 17:45:00  1.500908 -2.575753 -0.664001 -0.869288
2020-11-09 18:45:00  0.234116 -0.921579  1.252780 -0.459613
2020-11-09 19:45:00 -1.581432 -0.234228 -1.344278 -0.826298
2020-11-09 20:45:00 -1.364293  1.200433  1.443988  0.809296
2020-11-09 21:45:00 -1.949003  1.290937  1.618971  1.624421
2020-11-09 22:45:00 -0.112020  1.669798  0.524994  1.621564
2020-11-09 23:45:00  1.020831 -0.910490 -0.000111 -0.400006
```

[22]:
```
df.sort_values(by="C", ascending=False)
```

[22]:
```
                            A         B         C         D
2020-11-09 22:45:00  1.621564  0.524994  1.669798 -0.112020
2020-11-09 21:45:00  1.624421  1.618971  1.290937 -1.949003
2020-11-09 20:45:00  0.809296  1.443988  1.200433 -1.364293
2020-11-09 14:45:00  0.379644  2.386360 -0.197379 -0.167745
2020-11-09 19:45:00 -0.826298 -1.344278 -0.234228 -1.581432
2020-11-09 15:45:00 -0.375580 -0.277857 -0.267869 -0.073870
2020-11-09 16:45:00  0.326133 -0.454830 -0.353803 -0.181075
2020-11-09 23:45:00 -0.400006 -0.000111 -0.910490  1.020831
2020-11-09 18:45:00 -0.459613  1.252780 -0.921579  0.234116
2020-11-09 17:45:00 -0.869288 -0.664001 -2.575753  1.500908
```

1.2.3 Selection

Slicing DataFrame slicing allows to select a subset of the DataFrame, or an entire column (a Series):

[23]:
```
# standard and safe
print(df['A'], '\n', type(df['A']), '\n') # Returns a Series (a column)

# equivalent but dangerous (imagine blank spaces in the name of the column, or␣
 ↪a column named "T")
print(df.A, '\n')
```

```
2020-11-09 14:45:00    0.379644
2020-11-09 15:45:00   -0.375580
2020-11-09 16:45:00    0.326133
2020-11-09 17:45:00   -0.869288
2020-11-09 18:45:00   -0.459613
2020-11-09 19:45:00   -0.826298
2020-11-09 20:45:00    0.809296
2020-11-09 21:45:00    1.624421
2020-11-09 22:45:00    1.621564
2020-11-09 23:45:00   -0.400006
Freq: H, Name: A, dtype: float64
 <class 'pandas.core.series.Series'>

2020-11-09 14:45:00    0.379644
2020-11-09 15:45:00   -0.375580
2020-11-09 16:45:00    0.326133
2020-11-09 17:45:00   -0.869288
2020-11-09 18:45:00   -0.459613
2020-11-09 19:45:00   -0.826298
2020-11-09 20:45:00    0.809296
2020-11-09 21:45:00    1.624421
2020-11-09 22:45:00    1.621564
2020-11-09 23:45:00   -0.400006
Freq: H, Name: A, dtype: float64
```

Numpy-like slicing by row ranges is possible, and usually returns a **view** of the original DataFrame:

[24]:
```
# selecting rows by range. Returns another DataFrame (usually a view)
print(df[0:3], '\n')

# or by index range
print(df["2020-11-09 14:45:00" : "2020-11-09 16:45:00"])
```

```
                           A         B         C         D
2020-11-09 14:45:00  0.379644  2.386360 -0.197379 -0.167745
2020-11-09 15:45:00 -0.375580 -0.277857 -0.267869 -0.073870
```

```
2020-11-09 16:45:00  0.326133 -0.454830 -0.353803 -0.181075
```

```
                            A         B         C         D
2020-11-09 14:45:00  0.379644  2.386360 -0.197379 -0.167745
2020-11-09 15:45:00 -0.375580 -0.277857 -0.267869 -0.073870
2020-11-09 16:45:00  0.326133 -0.454830 -0.353803 -0.181075
```

Selection by label The most common way to select elements, rows, or columns in a DataFrame is by using the `.loc[]` method.

`.loc` supports multi-indexing, and usually returns a **copy** of the DataFrame.

[25]:
```
# getting a part of the DataFrame (in this case, a row)) using a label. Returns␣
↪a Series
dfs = df.loc[dates[0]] # equivalent to df.loc[df.index[0]]
print(dfs, '\n', type(dfs), '\n')
```

```
A    0.379644
B    2.386360
C   -0.197379
D   -0.167745
Name: 2020-11-09 14:45:00, dtype: float64
 <class 'pandas.core.series.Series'>
```

[26]:
```
# selecting on a multi-axis by label:
dfa = df.loc[:, ['A','B']]
dfa
```

[26]:
```
                            A         B
2020-11-09 14:45:00  0.379644  2.386360
2020-11-09 15:45:00 -0.375580 -0.277857
2020-11-09 16:45:00  0.326133 -0.454830
2020-11-09 17:45:00 -0.869288 -0.664001
2020-11-09 18:45:00 -0.459613  1.252780
2020-11-09 19:45:00 -0.826298 -1.344278
2020-11-09 20:45:00  0.809296  1.443988
2020-11-09 21:45:00  1.624421  1.618971
2020-11-09 22:45:00  1.621564  0.524994
2020-11-09 23:45:00 -0.400006 -0.000111
```

[27]:
```
# showing label slicing, both endpoints are included:
df.loc['2020-11-09 18:45:00':'2020-11-09 20:45:00', ['A','B']]
```

[27]:
```
                            A         B
2020-11-09 18:45:00 -0.459613  1.252780
2020-11-09 19:45:00 -0.826298 -1.344278
2020-11-09 20:45:00  0.809296  1.443988
```

[28]:
```
# getting an individual element
print(df.loc[dates[1], 'A'], '\n', type(df.loc[dates[1], 'A']), '\n')
```

```
-0.3755803170504277
 <class 'numpy.float64'>
```

The `.at()` method is equivalent to `.loc[]`. Use `at` if you only need to get or set a single value in a DataFrame or Series.

[29]:
```
print(df.at[dates[1], 'A'])
```

```
-0.3755803170504277
```

Selecting by position `.iloc[]` is similar to `.loc[]`, but instead of labels, it uses pure integer-location based indexing for selection by position.

But differently from `.loc[]`, `.iloc[]` usually returns a **view**, not a copy.

[30]:
```
# select via the position of the passed integers:
print(df.iloc[3], '\n', type(df.iloc[3]), '\n')
```

```
A   -0.869288
B   -0.664001
C   -2.575753
D    1.500908
Name: 2020-11-09 17:45:00, dtype: float64
 <class 'pandas.core.series.Series'>
```

If you specify just one axis or idex, a Series is returned. If you specify both axis or indices, you get a DataFrame instead:

[31]:
```
# row and column ranges selected with numpy-like notation:
dfv = df.iloc[3:5, 0:2]
print(dfv, '\n', type(df.iloc[3:5, 0:2]), '\n')
```

```
                           A         B
2020-11-09 17:45:00 -0.869288 -0.664001
2020-11-09 18:45:00 -0.459613  1.252780
 <class 'pandas.core.frame.DataFrame'>
```

[32]:
```
# selection of multiple elements with lists
df.iloc[[1, 2, 4], [0, 2]] # selecting rows 1,2 and 4 for columns 0 and 2
```

[32]:
```
                           A         C
2020-11-09 15:45:00 -0.375580 -0.267869
2020-11-09 16:45:00  0.326133 -0.353803
2020-11-09 18:45:00 -0.459613 -0.921579
```

[33]:
```
# slicing rows explicitly
df.iloc[1:3, :]

# slicing columns explicitly
df.iloc[:, 1:3]
```

[33]:

	B	C
2020-11-09 14:45:00	2.386360	-0.197379
2020-11-09 15:45:00	-0.277857	-0.267869
2020-11-09 16:45:00	-0.454830	-0.353803
2020-11-09 17:45:00	-0.664001	-2.575753
2020-11-09 18:45:00	1.252780	-0.921579
2020-11-09 19:45:00	-1.344278	-0.234228
2020-11-09 20:45:00	1.443988	1.200433
2020-11-09 21:45:00	1.618971	1.290937
2020-11-09 22:45:00	0.524994	1.669798
2020-11-09 23:45:00	-0.000111	-0.910490

Similary to `.loc[]` and `.at[]`, there is also `.iat[]` alongside `.iloc[]`:

[34]:
```
# selecting an individual element by position: no difference between iloc and␣
 ↪iat
print(df.iloc[1,1], ", type:", type(df.iloc[1,1]))
print(df.iat[1,1], ", type:", type(df.iat[1,1]))
```

```
-0.2778572510870173 , type: <class 'numpy.float64'>
-0.2778572510870173 , type: <class 'numpy.float64'>
```

Masks Boolean masks can be used in the same way as Numpy, and they represent a very powerful way of filtering out data with certain features. Just like Numpy fancy indexing, using a mask usually returns a **copy** of the DataFrame.

[35]:
```
# Selecting on the basis of boolean conditions applied to the whole DataFrame
dfc = df[df > 0]
dfc.iat[0, 0] = -99
# a DataFrame with the same shape is returned, with NaN's where condition is␣
 ↪not met
# Note that when a NaN is present in a column of integers, the column (Series)␣
 ↪is casted to float
dfc
```

[35]:

	A	B	C	D
2020-11-09 14:45:00	-99.000000	2.386360	NaN	NaN
2020-11-09 15:45:00	NaN	NaN	NaN	NaN
2020-11-09 16:45:00	0.326133	NaN	NaN	NaN
2020-11-09 17:45:00	NaN	NaN	NaN	1.500908
2020-11-09 18:45:00	NaN	1.252780	NaN	0.234116

```
2020-11-09 19:45:00       NaN       NaN       NaN       NaN
2020-11-09 20:45:00  0.809296  1.443988  1.200433       NaN
2020-11-09 21:45:00  1.624421  1.618971  1.290937       NaN
2020-11-09 22:45:00  1.621564  0.524994  1.669798       NaN
2020-11-09 23:45:00       NaN       NaN       NaN  1.020831
```

[36]:
```
# Filter by a boolean condition on the values of a single column
mask = dfc['B'] < 0.5
mask
```

[36]:
```
2020-11-09 14:45:00    False
2020-11-09 15:45:00    False
2020-11-09 16:45:00    False
2020-11-09 17:45:00    False
2020-11-09 18:45:00    False
2020-11-09 19:45:00    False
2020-11-09 20:45:00    False
2020-11-09 21:45:00    False
2020-11-09 22:45:00    False
2020-11-09 23:45:00    False
Freq: H, Name: B, dtype: bool
```

[37]:
```
# Filter only the rows that correspond to a True in the Series used as mask
dfc[mask]
```

[37]:
```
Empty DataFrame
Columns: [A, B, C, D]
Index: []
```

Queries

Pandas uses a database-like engine to select elements according to a query on the columns of the DataFrame:

[38]:
```
dfq = df.query('C > 0.5')
dfq
```

[38]:
```
                            A         B         C         D
2020-11-09 20:45:00  0.809296  1.443988  1.200433 -1.364293
2020-11-09 21:45:00  1.624421  1.618971  1.290937 -1.949003
2020-11-09 22:45:00  1.621564  0.524994  1.669798 -0.112020
```

which is equivalent to `dfq = df[df['C'] > 0.5]`:

[39]:
```
dfw = df[df['C'] > 0.5]
dfw
```

[39]:
```
                            A         B         C         D
2020-11-09 20:45:00  0.809296  1.443988  1.200433 -1.364293
2020-11-09 21:45:00  1.624421  1.618971  1.290937 -1.949003
2020-11-09 22:45:00  1.621564  0.524994  1.669798 -0.112020
```

1.2.4 Copy and views in DataFrames

The view/copy behaviour in Pandas is not as easy as it may appear, as there are counter-intuitive exceptions. There was a plan to fix this by quite some time, but a fix has not been deployed yet.

Check this discussion here:

```
In numpy, the rules for when you get views and when you don't are a little complicated, but the

But in pandas, whether you get a view or not-and whether changes made to a view will propagate
depends on the structure and data types in the original DataFrame.
```

In summary, there is only one way to write safe code when dealing with slides of a dataframe: after every instruction that selects a subset of a DataFrame, force the copy by appending `.copy()` to the slice.

1.2.5 Assignement

Assignment is typically performed after a selection:

[40]:
```
# Make sure to copy the DataFrame if you plan to modify it, and you don't want␣
 ↪to change the original object
dfa = df.copy()

# setting values by label (same as by position)
dfa.at[dates[0], 'A'] = -1

# setting and assigning a numpy array
dfa['D'] = np.array([5] * len(dfa))

# defining a brand new column by means of a pd.Series: indexes must be the same!
dfa['E'] = pd.Series(np.arange(len(dfa))*2, index=dfa.index)

# using masks for assigment
dfa[dfa < 0] = -dfa

dfa
```

[40]:
```
                            A         B         C  D   E
2020-11-09 14:45:00  1.000000  2.386360  0.197379  5   0
2020-11-09 15:45:00  0.375580  0.277857  0.267869  5   2
2020-11-09 16:45:00  0.326133  0.454830  0.353803  5   4
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5   6
2020-11-09 18:45:00  0.459613  1.252780  0.921579  5   8
```

```
2020-11-09 19:45:00  0.826298  1.344278  0.234228  5  10
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5  12
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5  14
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5  16
2020-11-09 23:45:00  0.400006  0.000111  0.910490  5  18
```

1.2.6 Dropping

Dropping columns is an example of **a method that does not modify the original object**, and returns a new modified object. In other words, if you want to keep the modified DataFrame, perform a new assignment:

`df = df.drop(...)`

Alternatively, the modification of the original object can be forced by specifying `inplace=True` among the arguments.

[41]:
```
dfb = dfa.copy()

# Dropping by column..
dfb.drop(['E'], axis=1)
dfb
```

[41]:
```
                            A         B         C  D   E
2020-11-09 14:45:00  1.000000  2.386360  0.197379  5   0
2020-11-09 15:45:00  0.375580  0.277857  0.267869  5   2
2020-11-09 16:45:00  0.326133  0.454830  0.353803  5   4
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5   6
2020-11-09 18:45:00  0.459613  1.252780  0.921579  5   8
2020-11-09 19:45:00  0.826298  1.344278  0.234228  5  10
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5  12
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5  14
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5  16
2020-11-09 23:45:00  0.400006  0.000111  0.910490  5  18
```

As you can see, there is no effect on the original object. That's because new object is returned instead. To keep it, there are two alternatives:

[42]:
```
dfc = dfb.drop(columns=['E'])
dfc
```

[42]:
```
                            A         B         C  D
2020-11-09 14:45:00  1.000000  2.386360  0.197379  5
2020-11-09 15:45:00  0.375580  0.277857  0.267869  5
2020-11-09 16:45:00  0.326133  0.454830  0.353803  5
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5
2020-11-09 18:45:00  0.459613  1.252780  0.921579  5
2020-11-09 19:45:00  0.826298  1.344278  0.234228  5
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5
```

```
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5
2020-11-09 23:45:00  0.400006  0.000111  0.910490  5
```

[43]:
```
dfb.drop(columns=['E'], inplace=True) # equivalent to the previous one, but the␣
↪original object has been replace inplace
dfb
```

[43]:
```
                            A         B         C  D
2020-11-09 14:45:00  1.000000  2.386360  0.197379  5
2020-11-09 15:45:00  0.375580  0.277857  0.267869  5
2020-11-09 16:45:00  0.326133  0.454830  0.353803  5
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5
2020-11-09 18:45:00  0.459613  1.252780  0.921579  5
2020-11-09 19:45:00  0.826298  1.344278  0.234228  5
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5
2020-11-09 23:45:00  0.400006  0.000111  0.910490  5
```

1.2.7 Dealing with missing data

Pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. If there is a `NaN` entry in a Series of integers, the type of the Series will be changed to floats.

[44]:
```
df_wNan = dfb[dfb > 0.5]
df_wNan
```

[44]:
```
                            A         B         C  D
2020-11-09 14:45:00  1.000000  2.386360       NaN  5
2020-11-09 15:45:00       NaN       NaN       NaN  5
2020-11-09 16:45:00       NaN       NaN       NaN  5
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5
2020-11-09 18:45:00       NaN  1.252780  0.921579  5
2020-11-09 19:45:00  0.826298  1.344278       NaN  5
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5
2020-11-09 23:45:00       NaN       NaN  0.910490  5
```

[45]:
```
# dropping rows with at least a Nan
df_wNan.dropna(how='any')
df_wNan
```

[45]:
```
                            A         B         C  D
2020-11-09 14:45:00  1.000000  2.386360       NaN  5
```

```
2020-11-09 15:45:00       NaN       NaN       NaN  5
2020-11-09 16:45:00       NaN       NaN       NaN  5
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5
2020-11-09 18:45:00       NaN  1.252780  0.921579  5
2020-11-09 19:45:00  0.826298  1.344278       NaN  5
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5
2020-11-09 23:45:00       NaN       NaN  0.910490  5
```

[46]:
```
# getting a mask
df_wNan.isna()
# df_wNan.notna()
```

[46]:
```
                        A      B      C      D
2020-11-09 14:45:00  False  False   True  False
2020-11-09 15:45:00   True   True   True  False
2020-11-09 16:45:00   True   True   True  False
2020-11-09 17:45:00  False  False  False  False
2020-11-09 18:45:00   True  False  False  False
2020-11-09 19:45:00  False  False   True  False
2020-11-09 20:45:00  False  False  False  False
2020-11-09 21:45:00  False  False  False  False
2020-11-09 22:45:00  False  False  False  False
2020-11-09 23:45:00   True   True  False  False
```

[47]:
```
# filling missing data (not recommended, unless you really mean it)
df_wNan.fillna(value=0)
```

[47]:
```
                           A         B         C  D
2020-11-09 14:45:00  1.000000  2.386360  0.000000  5
2020-11-09 15:45:00  0.000000  0.000000  0.000000  5
2020-11-09 16:45:00  0.000000  0.000000  0.000000  5
2020-11-09 17:45:00  0.869288  0.664001  2.575753  5
2020-11-09 18:45:00  0.000000  1.252780  0.921579  5
2020-11-09 19:45:00  0.826298  1.344278  0.000000  5
2020-11-09 20:45:00  0.809296  1.443988  1.200433  5
2020-11-09 21:45:00  1.624421  1.618971  1.290937  5
2020-11-09 22:45:00  1.621564  0.524994  1.669798  5
2020-11-09 23:45:00  0.000000  0.000000  0.910490  5
```

1.2.8 Operations

Operations on the elements of a DataFrame are quite straightforward, as the syntax is the same as the one used for Series. Also for DataFrames, operations are performed between elements that share the same labels. Operations on columns are extremly fast, almost as fast as the actual operation between elements in a row.

[48]:
```
# Some statistics (mean() just as an example)
# on rows
print(df.mean(axis=0), '\n')
# on columns
print(df.mean(axis=1), '\n')
```

```
A    0.183027
B    0.448602
C   -0.129993
D   -0.267358
dtype: float64

2020-11-09 14:45:00    0.600220
2020-11-09 15:45:00   -0.248794
2020-11-09 16:45:00   -0.165894
2020-11-09 17:45:00   -0.652034
2020-11-09 18:45:00    0.026426
2020-11-09 19:45:00   -0.996559
2020-11-09 20:45:00    0.522356
2020-11-09 21:45:00    0.646332
2020-11-09 22:45:00    0.926084
2020-11-09 23:45:00   -0.072444
Freq: H, dtype: float64
```

[49]:
```
# Global operations on columns
df.apply(np.sum) # or whatever function defined by the user
```

[49]:
```
A    1.830271
B    4.486015
C   -1.299934
D   -2.673582
dtype: float64
```

[50]:
```
# Also lambda functions are accepted
df.apply(lambda x: x - x.max())
```

[50]:

	A	B	C	D
2020-11-09 14:45:00	-1.244777	0.000000	-1.867177	-1.668653
2020-11-09 15:45:00	-2.000001	-2.664218	-1.937667	-1.574778
2020-11-09 16:45:00	-1.298288	-2.841191	-2.023601	-1.681983
2020-11-09 17:45:00	-2.493710	-3.050361	-4.245551	0.000000
2020-11-09 18:45:00	-2.084034	-1.133581	-2.591377	-1.266791
2020-11-09 19:45:00	-2.450719	-3.730638	-1.904026	-3.082339
2020-11-09 20:45:00	-0.815125	-0.942372	-0.469365	-2.865201
2020-11-09 21:45:00	0.000000	-0.767390	-0.378861	-3.449911
2020-11-09 22:45:00	-0.002857	-1.861367	0.000000	-1.612927

```
2020-11-09 23:45:00 -2.024428 -2.386472 -2.580287 -0.480076
```

[51]:
```
# syntax is as usual similar to that of numpy arrays
df['S'] = df['A'] + df['C']
df
```

[51]:

	A	B	C	D	S
2020-11-09 14:45:00	0.379644	2.386360	-0.197379	-0.167745	0.182265
2020-11-09 15:45:00	-0.375580	-0.277857	-0.267869	-0.073870	-0.643450
2020-11-09 16:45:00	0.326133	-0.454830	-0.353803	-0.181075	-0.027671
2020-11-09 17:45:00	-0.869288	-0.664001	-2.575753	1.500908	-3.445042
2020-11-09 18:45:00	-0.459613	1.252780	-0.921579	0.234116	-1.381193
2020-11-09 19:45:00	-0.826298	-1.344278	-0.234228	-1.581432	-1.060526
2020-11-09 20:45:00	0.809296	1.443988	1.200433	-1.364293	2.009729
2020-11-09 21:45:00	1.624421	1.618971	1.290937	-1.949003	2.915358
2020-11-09 22:45:00	1.621564	0.524994	1.669798	-0.112020	3.291362
2020-11-09 23:45:00	-0.400006	-0.000111	-0.910490	1.020831	-1.310496

1.2.9 Application of a function: apply vs transform

User-defined or standard functions can be applied on entire DataFrames or columns, with very short execution times.

There are two main methods, `apply()` and `transform()`:

[52]:
```
def dcos(theta):
    theta = theta * (np.pi / 180)
    return np.cos(theta)

# Apply method with custom function
dfa['cosine'] = dfa["E"].apply(dcos)

# Transform method with lambda function
dfa['EplusOne'] = dfa["E"].transform(lambda x: x + 1)
dfa
```

[52]:

	A	B	C	D	E	cosine	EplusOne
2020-11-09 14:45:00	1.000000	2.386360	0.197379	5	0	1.000000	1
2020-11-09 15:45:00	0.375580	0.277857	0.267869	5	2	0.999391	3
2020-11-09 16:45:00	0.326133	0.454830	0.353803	5	4	0.997564	5
2020-11-09 17:45:00	0.869288	0.664001	2.575753	5	6	0.994522	7
2020-11-09 18:45:00	0.459613	1.252780	0.921579	5	8	0.990268	9
2020-11-09 19:45:00	0.826298	1.344278	0.234228	5	10	0.984808	11
2020-11-09 20:45:00	0.809296	1.443988	1.200433	5	12	0.978148	13
2020-11-09 21:45:00	1.624421	1.618971	1.290937	5	14	0.970296	15
2020-11-09 22:45:00	1.621564	0.524994	1.669798	5	16	0.961262	17
2020-11-09 23:45:00	0.400006	0.000111	0.910490	5	18	0.951057	19

The major differences between `apply` and `transform` are:

- Input: `apply` passes all the columns to the custom function, while `transform` passes each column.
- Output: the custom function passed to `apply` can return a scalar, or a Series or DataFrame, while the custom function passed to `transform` must return a sequence (a Series, array or list) with the same length.

In summary, `transform` works on just one Series, and `apply` works on the entire DataFrame.

1.2.10 Merge

Pandas provides various functions for easily combining together Series and DataFrames in join / merge-type operations.

Concat

Concatenation (adding rows) is straightforward:

[53]:
```
rdf = pd.DataFrame(np.arange(40).reshape(10, 4))
rdf
```

[53]:

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19
5	20	21	22	23
6	24	25	26	27
7	28	29	30	31
8	32	33	34	35
9	36	37	38	39

[54]:
```
# split DataFrame into 3 pieces, row-wise
pieces = [rdf[:3], rdf[3:7], rdf[7:]]
print(pieces, '\n')
pieces[2]
```

```
[   0  1   2   3
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11,     0   1   2   3
3  12  13  14  15
4  16  17  18  19
5  20  21  22  23
6  24  25  26  27,     0   1   2   3
7  28  29  30  31
8  32  33  34  35
```

```
9  36  37  38  39]
```

[54]:
```
     0   1   2   3
7   28  29  30  31
8   32  33  34  35
9   36  37  38  39
```

[55]:
```
# put it back together
pd.concat(pieces)

# in this case, indices are already set; if they are not, indices can be ignored
#pd.concat(pieces, ignore_index=True)
```

[55]:
```
     0   1   2   3
0    0   1   2   3
1    4   5   6   7
2    8   9  10  11
3   12  13  14  15
4   16  17  18  19
5   20  21  22  23
6   24  25  26  27
7   28  29  30  31
8   32  33  34  35
9   36  37  38  39
```

In case of dimension mismatch, `NaN` are added where needed.

Merge/Join

SQL-like operations on table can be performed on DataFrames. This is a quite advanced use case, refer to the doc for more info/examples.

[56]:
```
left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})

pd.merge(left, right, on="key")
```

[56]:
```
   key  lval  rval
0  foo     1     4
1  bar     2     5
```

1.2.11 Grouping

In real world applications, it's quite common that several entries (row) belong to a certain entity, or "group". DataFrames have a powerful tool to perform operations on entries of the same group. The method is called `.groupby()`, and it usually involves one or more of the following steps:

- Splitting the data into groups based on some criteria
- Applying a function to each group independently

- Combining the results into a data structure

[57]:
```
gdf = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                           'foo', 'bar', 'foo', 'foo'],
                    'B' : [1, 1, 2, 3, 2, 2, 1, 3],
                    'C' : np.arange(8),
                    'D' : np.linspace(10, -10, 8)})
gdf
```

[57]:

	A	B	C	D
0	foo	1	0	10.000000
1	bar	1	1	7.142857
2	foo	2	2	4.285714
3	bar	3	3	1.428571
4	foo	2	4	-1.428571
5	bar	2	5	-4.285714
6	foo	1	6	-7.142857
7	foo	3	7	-10.000000

[58]:
```
# Grouping and then applying the sum()
# function to the resulting groups (effective only where numerical values are␣
 ↪present)
gdf.groupby('A').sum()
```

[58]:

A	B	C	D
bar	6	9	4.285714
foo	9	19	-4.285714

[59]:
```
# Example: find maximum value in column D for each group, and assign the value␣
 ↪to a new column
gdf['M'] = gdf.groupby('A')['D'].transform(np.max)
gdf
```

```
/tmp/ipykernel_4223/3525720408.py:2: FutureWarning: The provided callable
<function max at 0x7f41e822c550> is currently using SeriesGroupBy.max. In a
future version of pandas, the provided callable will be used directly. To keep
current behavior pass the string "max" instead.
  gdf['M'] = gdf.groupby('A')['D'].transform(np.max)
```

[59]:

	A	B	C	D	M
0	foo	1	0	10.000000	10.000000
1	bar	1	1	7.142857	7.142857
2	foo	2	2	4.285714	10.000000
3	bar	3	3	1.428571	7.142857
4	foo	2	4	-1.428571	10.000000
5	bar	2	5	-4.285714	7.142857
6	foo	1	6	-7.142857	10.000000

```
7  foo  3  7 -10.000000  10.000000
```

1.2.12 Multi-indexing

Hierarchical / Multi-level indexing allows sophisticated data analysis on higher dimensional data. In practice, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like Series (1D) and DataFrames (2D).

[60]:
```
# Creat multi-dimensional index
tuples = list(zip(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
          ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']))
multi_index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
print(multi_index, '\n', type(multi_index), '\n')

# Create multi-indexed dataframe or series
s = pd.Series(np.arange(8)/np.pi, index=multi_index)
s
```

```
MultiIndex([('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two'),
            ('foo', 'one'),
            ('foo', 'two'),
            ('qux', 'one'),
            ('qux', 'two')],
           names=['first', 'second'])
 <class 'pandas.core.indexes.multi.MultiIndex'>
```

[60]:
```
first  second
bar    one       0.000000
       two       0.318310
baz    one       0.636620
       two       0.954930
foo    one       1.273240
       two       1.591549
qux    one       1.909859
       two       2.228169
dtype: float64
```

[61]:
```
# multi-indexing enables further features of the groupby method,
# e.g. when group-by by multiple columns
gdf.groupby(['A', 'B']).sum()
```

[61]:
```
          C         D         M
A   B
```

```
bar 1  1   7.142857   7.142857
    2  5  -4.285714   7.142857
    3  3   1.428571   7.142857
foo 1  6   2.857143  20.000000
    2  6   2.857143  20.000000
    3  7 -10.000000  10.000000
```

1.3 Summary: a demonstration of the efficiency of the DataFrame

Let's go the hard way and load a (relatively) large dataset with approximately 1 million rows:

[64]:
```
# Uncomment to download the file. Run the command just once
!wget https://www.dropbox.com/s/xvjzaxzz3ysphme/data_000637.txt -P ./data/
```

```
--2023-11-15 10:38:32--
https://www.dropbox.com/s/xvjzaxzz3ysphme/data_000637.txt
Resolving www.dropbox.com (www.dropbox.com)… 162.125.69.18,
2620:100:6025:18::a27d:4512
Connecting to www.dropbox.com (www.dropbox.com)|162.125.69.18|:443… connected.
HTTP request sent, awaiting response… 302 Found
Location: /s/raw/xvjzaxzz3ysphme/data_000637.txt [following]
--2023-11-15 10:38:32--
https://www.dropbox.com/s/raw/xvjzaxzz3ysphme/data_000637.txt
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response… 302 Found
Location: https://uc86616997f35d3378bd34c2af8e.dl.dropboxusercontent.com/cd/0/in
line/CHkGjOjbZN04N7b1J1e-
AjMDYMcfcQi7XrkevAnxhyvfcmHnlHWYEXnHwrd7kmUmtRotkVif0SuB0_qMAEQhVQfA52yh7ZZBXnNa
HCiIK-3wiKhfQ2b5pnXNIllPw39gSus/file# [following]
--2023-11-15 10:38:32--  https://uc86616997f35d3378bd34c2af8e.dl.dropboxusercont
ent.com/cd/0/inline/CHkGjOjbZN04N7b1J1e-
AjMDYMcfcQi7XrkevAnxhyvfcmHnlHWYEXnHwrd7kmUmtRotkVif0SuB0_qMAEQhVQfA52yh7ZZBXnNa
HCiIK-3wiKhfQ2b5pnXNIllPw39gSus/file
Resolving uc86616997f35d3378bd34c2af8e.dl.dropboxusercontent.com
(uc86616997f35d3378bd34c2af8e.dl.dropboxusercontent.com)… 162.125.69.15,
2620:100:6025:15::a27d:450f
Connecting to uc86616997f35d3378bd34c2af8e.dl.dropboxusercontent.com
(uc86616997f35d3378bd34c2af8e.dl.dropboxusercontent.com)|162.125.69.15|:443…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 33179236 (32M) [text/plain]
Saving to: ‘./data/data_000637.txt’

data_000637.txt     100%[===================>]  31,64M  10,1MB/s    in 3,1s

2023-11-15 10:38:36 (10,1 MB/s) - ‘./data/data_000637.txt’ saved
[33179236/33179236]
```

[65]:
```
file_name = "./data/data_000637.txt"
data = pd.read_csv(file_name)
data
```

[65]:

	HEAD	FPGA	TDC_CHANNEL	ORBIT_CNT	BX_COUNTER	TDC_MEAS
0	1	0	123	3869200167	2374	26
1	1	0	124	3869200167	2374	27
2	1	0	63	3869200167	2553	28
3	1	0	64	3869200167	2558	19
4	1	0	64	3869200167	2760	25
…	…	…	…	…	…	…
1310715	1	0	62	3869211171	762	14
1310716	1	1	4	3869211171	763	11
1310717	1	0	64	3869211171	764	0
1310718	1	0	139	3869211171	769	0
1310719	1	0	61	3869211171	762	18

[1310720 rows x 6 columns]

Let's now do some operations among (elements of) columns

[66]:
```
itime = dt.datetime.now()
print("Begin time:", itime)

# the one-liner command
data['WEIGHTEDSUM'] = data['TDC_CHANNEL'] * 2.1 + data['BX_COUNTER'] * 0.1 + 2

ftime = dt.datetime.now()
print("End time:", ftime)
print("Elapsed time:", ftime - itime)

data
```

```
Begin time: 2023-11-15 10:38:45.322470
End time: 2023-11-15 10:38:45.346536
Elapsed time: 0:00:00.024066
```

[66]:

	HEAD	FPGA	TDC_CHANNEL	ORBIT_CNT	BX_COUNTER	TDC_MEAS	\
0	1	0	123	3869200167	2374	26	
1	1	0	124	3869200167	2374	27	
2	1	0	63	3869200167	2553	28	
3	1	0	64	3869200167	2558	19	
4	1	0	64	3869200167	2760	25	
…	…	…	…	…	…	…	
1310715	1	0	62	3869211171	762	14	
1310716	1	1	4	3869211171	763	11	
1310717	1	0	64	3869211171	764	0	
1310718	1	0	139	3869211171	769	0	

```
1310719     1     0           61   3869211171         762        18

         WEIGHTEDSUM
0              497.7
1              499.8
2              389.6
3              392.2
4              412.4
…                …
1310715        208.4
1310716         86.7
1310717        212.8
1310718        370.8
1310719        206.3

[1310720 rows x 7 columns]
```

[ ]:
```
# the loop
def conversion(data):
    result = []
    for i in range(len(data)):
        result.append(data.loc[data.index[i], 'TDC_CHANNEL'] * 2.1 + data.
 ↪loc[data.index[i], 'BX_COUNTER'] * 0.1 + 2)
    return result

itime = dt.datetime.now()
print("Begin time:", itime)
data['WEIGHTEDSUM'] = conversion(data)
ftime = dt.datetime.now()
print("End time:", ftime)
print("Elapsed time:", ftime - itime)

data
```

```
Begin time: 2023-11-15 10:38:49.141505
```

[ ]:
```

```