Data Analysis - Introduction to Pandas

Gabriele Pompa
gabriele.pompa@unisi.com

March 30, 2020

Contents

Resources:

- *Python for Finance (2nd ed.)*: Sec. 5.The DataFrame Class, 5.Basic Analytics, 5.Basic Visualization, 5.The Series Class, 5.Complex Selection, 5.Concatenation, Joining, and Merging, 5.Performance Aspects.
- *Pandas - Intro to data structures* (Series; DataFrame). From *Pandas - Getting started tutorials:*
  - What kind of data does pandas handle?,
  - *How do I select a subset of a `DataFrame`?*,
  - *How to create plots in pandas?*,
  - *How to create new columns derived from existing columns*,
  - *How to calculate summary statistics?*,
  - *How to combine data from multiple tables?*

Executive Summary

Pandas is a Python package providing you flexible, intuitive and powerful data-structures which allow you to work easily with spreadsheet-like relational data. Pandas defines two primary data-structures: Series and DataFrames.

The following sections are organized as follows:

- In Sec. 1 we introduce Pandas Series, which are 1-dimensional data-strucutures providing a great way to model time-series. In particular,
  - in Sec. 1.1 we show how to create Series using the `pd.Series()` constructor;
  - in Sec. 1.2 we show how to make basic plots using Series built-in methods;
  - in Sec. 1.3 we show how to select values using the `[]` access operator according to their indexes or conditional expressions;
  - in Sec. 1.4 we show how to compute basic analytics on Pandas Series;
  - in Sec. 1.5 we show how the alignment of data is handled by Pandas to combine two possibly unaligned Series;
  - finally, in Sec. 1.6, we make an excursus and show how to compute linear and logarithmic returns from a time-series of log-normal i.i.d. values.
- In Sec. 2 we introduce Pandas DataFrames, which are multi-dimensional data-structures providing great functionalities to work with spreadsheet like tables of data. In particular,
  - in Sec. 2.1 we show how to create DataFrames using the `pd.DataFrame()` constructor;
  - in Sec. 2.2 we show how to make basic plots using DataFrames built-in methods;
  - in Sec. 2.3 we show how to select columns using the `[]` access operator and how to select values according to their index and column labels as well as positional indexes using the `.loc[]` and `.iloc[]` access operators, respectively;
  - in Sec. 2.4 we show how to create and delete columns of a DataFrame;
  - in Sec. 2.5 we show how to compute basic analytics on Pandas DataFrames on a column- and row-wise base as well as on categorical groups of rows using the `.groupby()` method;

- in Sec. 2.6 we show how the alignment of data is handled by Pandas to combine two possibly unaligned DataFrames;
- in Sec. 2.7, we show how to concatenate and join two DataFrames, in common real life situations, using the `.join()` method and the `pd.merge()` function.

These are the basic imports that we need to work with NumPy, Pandas and to plot data using Matplotlib functionalities

[1]:
```python
# for NumPy arrays
import numpy as np

# for Pandas Series and DataFrame
import pandas as pd

# for Matplotlib plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

1 Pandas Series

A Pandas Series is a one-dimensional labeled array capable of holding any data type (Integers, Strings, Floats, etc.).

We can see it as a 1-dim NumPy array with an enhanced indexing.

1.1 Creation: `pd.Series()`

Series can be created using the constructor:

`pd.Series(data[, index, name])`

where:

- `data`: is the data content of theSeries. It can be a Python Dict, a NumPy 1-dim array or a scalar value (like 17).
- `index`: (optional) is the index of theSeries. It can be an array-like structure (e.g. a List) of the length of `data`. If not provided, default is `[0,1,...,len(data)-1]`.
- `name`: (optional) is a `str` representing the name of theSeries.

Here we consider the creation of a PandasSeries from a NumPy array. We refer to Intro to data structures - Series for other creational paradigms and full details.

So, let's define a simple vector

[2]:
```python
arr = np.linspace(0.0, 1.0, 11)
arr
```

[2]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

we pass the vector `arr` as the `data` parameter of `pd.Series()`

[3]:
```
s = pd.Series(data=arr)
s
```

[3]:
```
0     0.0
1     0.1
2     0.2
3     0.3
4     0.4
5     0.5
6     0.6
7     0.7
8     0.8
9     0.9
10    1.0
dtype: float64
```

The function `pd.Series()` returns a PandasSeries object. Each element `0.0, 0.1, ..., 1.0` is linked to its corresponding index. Notice that the `index` which is generated by default (since we didn't provide one explicitly) is `0, 1,..., 10 = len(arr)-1`.

[4]:
```
type(s)
```

[4]:
```
pandas.core.series.Series
```

Notice that the explicit assignment `data=arr` is optional and equivalent to `pd.Series(arr)`.

Similarly to NumPy arrays, PandasSeries also have meta-informative attributes. Let's have a look at some of them.

Similarly to NumPy arrays, the number of elements is given by

[5]:
```
s.size
```

[5]:
```
11
```

the data-type of the `data` stored

[6]:
```
s.dtype
```

[6]:
```
dtype('float64')
```

and, differently from arrays, you can directly access the index sequence:

[7]:
```
s.index
```

[7]:
```
RangeIndex(start=0, stop=11, step=1)
```

`RangeIndex` is the kind of `[0,1,...,len(data)-1]` index which Pandas creates by default when you don't input one explicitly.

You can give a name to theSeries, which is stored in the `.name` attribute of theSeries

[8]:
```
s.name = "Dummy Series"
s
```

[8]:
```
0     0.0
1     0.1
2     0.2
3     0.3
4     0.4
5     0.5
6     0.6
7     0.7
8     0.8
9     0.9
10    1.0
Name: Dummy Series, dtype: float64
```

If you want just the values (without the indexing) - that is, the original NumPy `arr` in our case - these can be accessed through the `.values` attribute

[9]:
```
s.values
```

[9]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

1.1.1 Time indexes: `pd.date_range()` and `pd.to_datetime()`

PandasSeries are the usual way to represent financial (and non-financial) time-series, which are sequences of values (prices, returns, spreads,…) indexed by a time index (calendar days, business days, etc.).

Pandas has a built-in constructor for time-indexes, which is `pd.date_range()`, which can be passed as the `index` parameter to `pd.Series()`.

Here we create a range of business days (denoted by the *frequency* `freq='B'`) starting from Jan 1st 2020. The range lasts a number of `periods` equal to the size of the number of data that we need to index (`arr.size`)

[10]:
```
dates = pd.date_range('2020-01-01', periods=arr.size, freq='B')
dates
```

[10]:
```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
               '2020-01-13', '2020-01-14', '2020-01-15'],
              dtype='datetime64[ns]', freq='B')
```

The kind of time-index that is returned is called a `DatetimeIndex`. As said, we can use the defined index in theSeries definition

[11]:
```python
s = pd.Series(data=arr, index=dates, name="Dummy Series")
s
```

[11]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

as seen before, the index is stored as the `.index` attribute of theSeries

[12]:
```python
s.index
```

[12]:
```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
               '2020-01-13', '2020-01-14', '2020-01-15'],
              dtype='datetime64[ns]', freq='B')
```

Now another function, which is appropriate to describe here, that will be useful in the indexing of a Series indexed by dates: `pd.to_datetime()`.

Function `pd.to_datetime()` can take in input a List of Strings representing dates and transform it in a `DatetimeIndex`, which can then be used to index aSeries and to access to specific rows (see later section).

[13]:
```python
listOfDatesStr = ['2020-01-01', '2020-01-13', '2020-01-15']
pd.to_datetime(listOfDatesStr)
```

[13]:
```
DatetimeIndex(['2020-01-01', '2020-01-13', '2020-01-15'],
dtype='datetime64[ns]', freq=None)
```

[14]:
```python
s_other = pd.Series(data=[0, 3, 4],
                    index=pd.to_datetime(listOfDatesStr),
                    name="pd.to_datetime() exampleSeries")
s_other
```

[14]:
```
2020-01-01    0
2020-01-13    3
2020-01-15    4
Name: pd.to_datetime() exampleSeries, dtype: int64
```

1.2 Basic plotting: `s.plot()` and `s.plot.bar()`

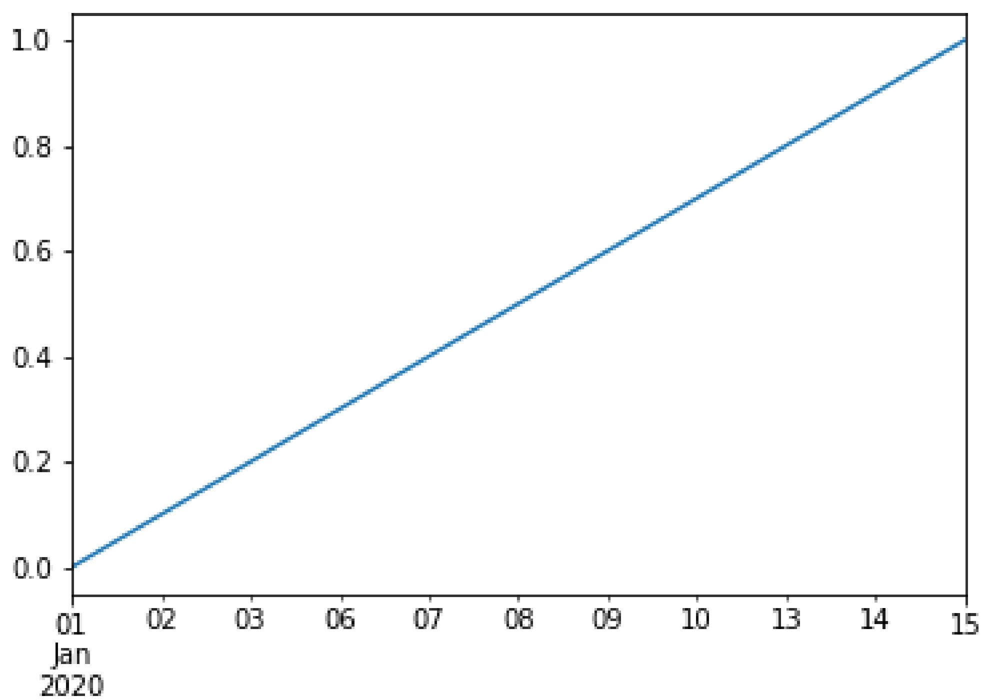
Plotting Series is as easy as it could be

[15]:
```
s
```

[15]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

[16]:
```
s.plot()
```

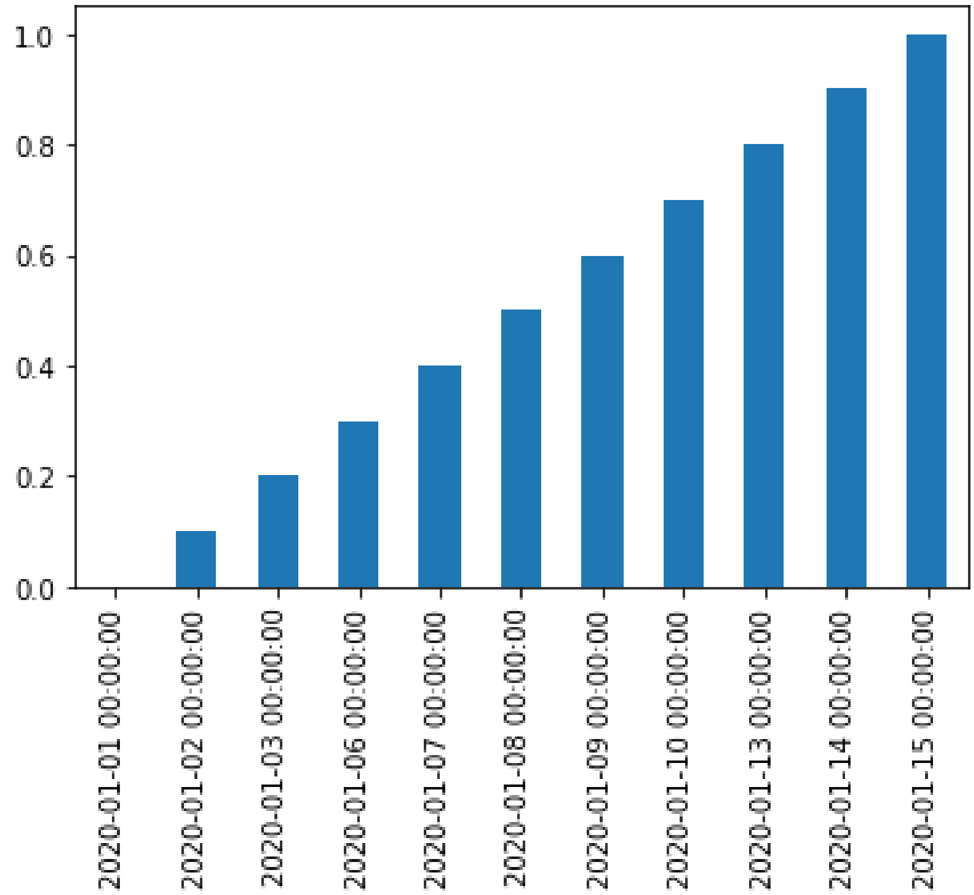
[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x19083c07048>
```



Besides the standard `.plot()`, there are tons of different plotting styles which can be called directly as a suffix of `.plot`. For example, a simple bar-plot of theSeries `s` can be drawn as

[17]:
```
s.plot.bar()
```

[17]: <matplotlib.axes._subplots.AxesSubplot at 0x19084128408>



1.3 Indexing and Slicing

Elements of a Pandas Series `s` can be accessed using the square brackets `[]` access operator as `s[]`. You can refer to elements or slices of theSeries according to the following ways of indexing:

	Indexing	Slicing
like a NumPy array	`s[i]`	`s[i:j:k]`
(example)	`s[2]`	`s[1:3:2]`
like a Python Dict	`s[indexLabel]`	`s[indexLabelStart:indexLabelEnd:k]`
(example)	`s['2020-01-01']`	`s['2020-01-01':'2020-01-07':2]`

or you can also pack together numeric indexes or index-labels inLists and get specific rows in output:

	syntax	example
withList of indexes	`s[[listOfNumericIndexes]]`	`s[[0,1,3]]`
withList of index labels	`s[[listOfIndexLabels]]`	`s[['a', 'c', 'd']]`
withList of dates labels	`s[pd.to_datetime([listOfDatesStr])]`	`s[pd.to_datetime(['2020-01-01', '2020-01-05'])]`

Notice that for index labels which are dates Strings, you have to use `pd.to_datetime()` to make them interpretable as valid (`DatetimeIndex`) indexes

[18]:
```
s
```

[18]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

You can rever to the second element as

[19]:
```
s[1]
```

[19]:
```
0.1
```

A single value is returned.

You can then slice theSeries as

[20]:
```
s[7:]
```

[20]:
```
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

and a `pd.Series` is returned. Notice that the index gets sliced too.

An example with the step parameter

[21]:
```
s[7::2] # from index 7 to the end, each two elements
```

[21]:
```
2020-01-10    0.7
2020-01-14    0.9
Freq: 2B, Name: Dummy Series, dtype: float64
```

Notice here, how the `Freq` description gets changed from 'B' (business-day) to '2B' (each two bd).

You can also use a List of numeric indexes to select some rows:

[22]:
```
s[[0,1,3,5]]
```

[22]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-06    0.3
2020-01-08    0.5
Name: Dummy Series, dtype: float64
```

Notices the double square brackets `[[...]]`. The innermost defines theList (e.g. `[0,1,3,5]`), whereas the outermost is the access operator `[]`.

Alternatively, you can slice `s` using the Strings that represent the labels `indexLabel` of the indexes. That is, as if theSeries was a Dict and the index labels the keys

[23]:
```
s['2020-01-08']
```

[23]:
```
0.5
```

and you can slice too using the index labels

[24]:
```
s['2020-01-08':'2020-01-13']
```

[24]:
```
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
Freq: B, Name: Dummy Series, dtype: float64
```

You can use the step parameter in the label-way of indexing too (notice the '2B' frequency as before)

[25]:
```
s['2020-01-08':'2020-01-13':2]
```

[25]:
```
2020-01-08    0.5
2020-01-10    0.7
Freq: 2B, Name: Dummy Series, dtype: float64
```

Similarly to Python Dicts, you can check whether an index is among theSeries indexes simply

[26]:
```
'2020-01-12' in s
```

[26]:
```
False
```

as one would expect a KeyError is raised if you try to select an element of theSeries using a label which is not an index

[27]:
```
# KeyError raised if you ask for a label that is not contained
# s['2020-01-12']
```

whereas if you use a label which is not an index in to slice theSeries - but that still is time-range of the DatetimeIndex index - well Pandas is smart enough to return you the relevant slice of theSeries anyway.

That is, this

[28]:
```
s['2020-01-12':]
```

[28]:
```
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

is equivalent to this

[29]:
```
s['2020-01-13':]
```

[29]:
```
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

1.3.1 Indexing with a List of dates using pd.to_datetime()

If you want to select just a few rows according to their dates labels, you can do it using pd.to_datetime() which makes theList of dates String interpretable as a valid indexer for theSeries

[30]:
```
s
```

[30]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

[31]:
```
pd.to_datetime(['2020-01-01', '2020-01-13', '2020-01-15'])
```

[31]:
```
DatetimeIndex(['2020-01-01', '2020-01-13', '2020-01-15'],
dtype='datetime64[ns]', freq=None)
```

[32]:
```python
s[pd.to_datetime(['2020-01-01', '2020-01-13', '2020-01-15'])]
```

[32]:
```
2020-01-01    0.0
2020-01-13    0.8
2020-01-15    1.0
Name: Dummy Series, dtype: float64
```

In particular, notice that if you input raw dates Strings in theSeries like

`s[['2020-01-01', '2020-01-13', '2020-01-15']]`

this will raise a `KeyError` because Pandas tries to look for raw Strings among the indexes, whereas what you actually meant were dates…

[33]:
```python
# raises KeyError
#
# s[['2020-01-01', '2020-01-13', '2020-01-15']]
```

Take-home message: when yourSeries is indexed by dates (a `DatetimeIndex` index) and you want to select a few rows using a List of dates Strings, like:

`['2020-01-01', '2020-01-13',...]`

just remember to convert them first through `pd.to_datetime()`, like

`pd.to_datetime(['2020-01-01', '2020-01-13',...])`

and then use to access rows of theSeries, like

`s[pd.to_datetime(['2020-01-01', '2020-01-13',...])]`

En passant, notice that if your Series is simply indexed by raw Strings, then you really can use them in a List indexing straight away. Quick example:

[34]:
```python
s_other = pd.Series(data=[1,3,5,7,9],
                    index=['a', 'b', 'c', 'd', 'e'],
                    name="Series indexed with raw Strings")

s_other
```

[34]:
```
a    1
b    3
c    5
d    7
e    9
Name: Series indexed with raw Strings, dtype: int64
```

[35]:
```python
s_other[['a', 'c', 'd']]
```

[35]:
```
a    1
c    5
d    7
Name: Series indexed with raw Strings, dtype: int64
```

1.3.2 Conditional Selection: filtering rows with Comparison and Logical operators

Comparison (`<`, `<=`, `>`, `>=`, `==`) and logical operators (`&` for logical *and*, `|` for logical *or*, `!` for logical *not*) work on wholeSeries at once. For example:

[36]:
```
s
```

[36]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

we can select the elements greater than a given threshold (0.5)

[37]:
```
s > 0.5
```

[37]:
```
2020-01-01    False
2020-01-02    False
2020-01-03    False
2020-01-06    False
2020-01-07    False
2020-01-08    False
2020-01-09     True
2020-01-10     True
2020-01-13     True
2020-01-14     True
2020-01-15     True
Freq: B, Name: Dummy Series, dtype: bool
```

which returns the sameSeries with original values substituted by boolean values, which can be used for index purposes:

[38]:
```
s[s > 0.5]
```

[38]:
```
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

Of course you can have also more complex conditional selections selections: let's ask for the slice of theSeries which is at most 0.2 and greater than 0.5

[39]:
```
s[(s <= 0.2) | (s > 0.5)]
```

[39]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Name: Dummy Series, dtype: float64
```

1.4 Basic Analytics

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

1.4.1 *Vectorized* operations

Series-Number and Series-Series operations are vectorized:

[40]:
```
s * 2
```

[40]:
```
2020-01-01    0.0
2020-01-02    0.2
2020-01-03    0.4
2020-01-06    0.6
2020-01-07    0.8
2020-01-08    1.0
2020-01-09    1.2
2020-01-10    1.4
2020-01-13    1.6
2020-01-14    1.8
2020-01-15    2.0
Freq: B, Name: Dummy Series, dtype: float64
```

[41]:
```
s + 10*s
```

[41]:
```
2020-01-01     0.0
2020-01-02     1.1
2020-01-03     2.2
2020-01-06     3.3
2020-01-07     4.4
2020-01-08     5.5
2020-01-09     6.6
2020-01-10     7.7
2020-01-13     8.8
2020-01-14     9.9
2020-01-15    11.0
Freq: B, Name: Dummy Series, dtype: float64
```

1.4.2 Built-in methods

There are tons of built-in methods

[42]:
```
s.sum()
```

[42]:
```
5.500000000000001
```

[43]:
```
s.cumsum()
```

[43]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.3
2020-01-06    0.6
2020-01-07    1.0
2020-01-08    1.5
2020-01-09    2.1
2020-01-10    2.8
2020-01-13    3.6
2020-01-14    4.5
2020-01-15    5.5
Freq: B, Name: Dummy Series, dtype: float64
```

1.4.3 Interoperability with NumPy's universal functions

Most of NumPy universal functions, which expect NumPy arrays in input, work with Pandas Series in input as well

[44]:
```
np.exp(s)
```

[44]:
```
2020-01-01    1.000000
2020-01-02    1.105171
2020-01-03    1.221403
2020-01-06    1.349859
2020-01-07    1.491825
```

```
2020-01-08    1.648721
2020-01-09    1.822119
2020-01-10    2.013753
2020-01-13    2.225541
2020-01-14    2.459603
2020-01-15    2.718282
Freq: B, Name: Dummy Series, dtype: float64
```

1.5 Data Alignment

NumPy arrays are all indexed by the same `0,1,...,len(array)-1` indexing. As we have seen Pandas Series offer the possibility to the user to define his or her own indexing. This means that you can have twoSeries which are *unaligned*. For example:

[45]:
```
s
```

[45]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

[46]:
```
s1 = s[s <= 0.6]
s1
```

[46]:
```
2020-01-01    0.0
2020-01-02    0.1
2020-01-03    0.2
2020-01-06    0.3
2020-01-07    0.4
2020-01-08    0.5
Freq: B, Name: Dummy Series, dtype: float64
```

[47]:
```
s2 = s[s >= 0.4]
s2
```

[47]:
```
2020-01-07    0.4
2020-01-08    0.5
2020-01-09    0.6
2020-01-10    0.7
```

```
2020-01-13    0.8
2020-01-14    0.9
2020-01-15    1.0
Freq: B, Name: Dummy Series, dtype: float64
```

This raises the question about how to define an operation such as `s1 + s2`

[48]:
```
s1 + s2
```

[48]:
```
2020-01-01    NaN
2020-01-02    NaN
2020-01-03    NaN
2020-01-06    NaN
2020-01-07    0.8
2020-01-08    1.0
2020-01-09    NaN
2020-01-10    NaN
2020-01-13    NaN
2020-01-14    NaN
2020-01-15    NaN
Freq: B, Name: Dummy Series, dtype: float64
```

well, given two unalignedSeries `s1` and `s2`, Pandas chose to index any combination of the two according to the *union* of their indexes (here `s1` has indexes from `2020-01-01` to `2020-01-08` whereas `s2` has indexes from `2020-01-07` to `2020-01-15`).

The `+` operator is applied to each element corresponding to indexes in common between the twoSeries (here indexes `2020-01-07` and `2020-01-08`). The rest of indexes (in the union of the indexes) are either not found in `s1` or in `s2`. Accordingly, their corresponding values in the `s1 + s2`Series will be marked as *missing* and denoted by `NaN` (which stands for Not-A-Number).

Of course this generalizes to any other operation:

[49]:
```
s3 = s1 * s2
s3
```

[49]:
```
2020-01-01     NaN
2020-01-02     NaN
2020-01-03     NaN
2020-01-06     NaN
2020-01-07    0.16
2020-01-08    0.25
2020-01-09     NaN
2020-01-10     NaN
2020-01-13     NaN
2020-01-14     NaN
2020-01-15     NaN
Freq: B, Name: Dummy Series, dtype: float64
```

What is cool (and really helps data analysis a lot) is that most basic analytics still works disregarding NaNs. That is, `NaN` are not counted.

[50]:
```
s3 ** 2
```

[50]:
```
2020-01-01       NaN
2020-01-02       NaN
2020-01-03       NaN
2020-01-06       NaN
2020-01-07    0.0256
2020-01-08    0.0625
2020-01-09       NaN
2020-01-10       NaN
2020-01-13       NaN
2020-01-14       NaN
2020-01-15       NaN
Freq: B, Name: Dummy Series, dtype: float64
```

[51]:
```
s3.sum()
```

[51]:
```
0.41000000000000003
```

[52]:
```
s3.mean()
```

[52]:
```
0.20500000000000002
```

[53]:
```
s3.std()
```

[53]:
```
0.06363961030678926
```

[54]:
```
s3.cumsum()
```

[54]:
```
2020-01-01     NaN
2020-01-02     NaN
2020-01-03     NaN
2020-01-06     NaN
2020-01-07    0.16
2020-01-08    0.41
2020-01-09     NaN
2020-01-10     NaN
2020-01-13     NaN
2020-01-14     NaN
2020-01-15     NaN
Freq: B, Name: Dummy Series, dtype: float64
```

1.6 *Excursus:* Returns time-series

A typical example of *unaligned* timeSeries is encountered when computing the time-series of the returns of an underlying time-series.

Let's simulate a time-series p_t as a time-series of i.i.d. (independent and identically distributed) log-normal random numbers.

Let's begin recalling the relation between normal and log-normal random variables. The random variable X is log-normally distributed if the logarithm $\ln(X)$ of X is normally distributed.

Therefore, if the variable $Y = \ln(X)$ is normally distributed with mean $E[Y] = \mu$ and variance $Var[Y] = \sigma^2$, that is

$$Y = \ln(X) \sim \mathcal{N}(\mu, \sigma^2)$$

then, $X = e^Y$ is log-normally distributed as

$$X = e^Y \sim \ln \mathcal{N}(\mu, \sigma^2)$$

with mean $E[X]$ and variance $Var[X]$ which are related to the first two moments of Y as follows (check Wikipedia):

$$E[X] = \exp\left(\mu + \frac{1}{2}\sigma^2\right)$$
$$Var[X] = \left[\exp\left(\sigma^2\right) - 1\right] \exp\left(2\mu + \sigma^2\right)$$

We shall use NumPy's `random.lognormal` function which expects in input the mean μ and standard-deviation σ of the underlying normal random variable Y and returns a NumPy array of i.i.d. log-normal random variables in output.

Warning Notice that this a very *elementary* time-series simulation. In particular, notice that this is **not** a simulation of a Geometric Brownian Motion (GBM). Intuitivelly, even we are sampling each p_t from a log-normal distribution, we are sampling them as i.i.d. random variables. Whereas, each subsequent number of a GBM, depends on the previous one. Stochastic process simulation will be covered in a future lesson.

[55]:
```
length = 20 # number of prices to simulate
```

[56]:
```
# we set the seed to have reproducible results. This has to be in the same
↪code-cell of the number extraction,
# otherwise Jupyter will forget it and reset it.
seed = np.random.seed(987654321)

mu, sigma = np.log(30.), np.log(1.1) # mean and std-dev of the underlying
↪normal random variable (\mu and \sigma above)

p = np.random.lognormal(mu, sigma, length)
```

and plot it using a business-day date range on the x-axis starting from Jan 1st to Jan 28th 2020 (we can use the `DatetimeIndex` returned by `pd.date_range()` in the Matplotlib plot function)

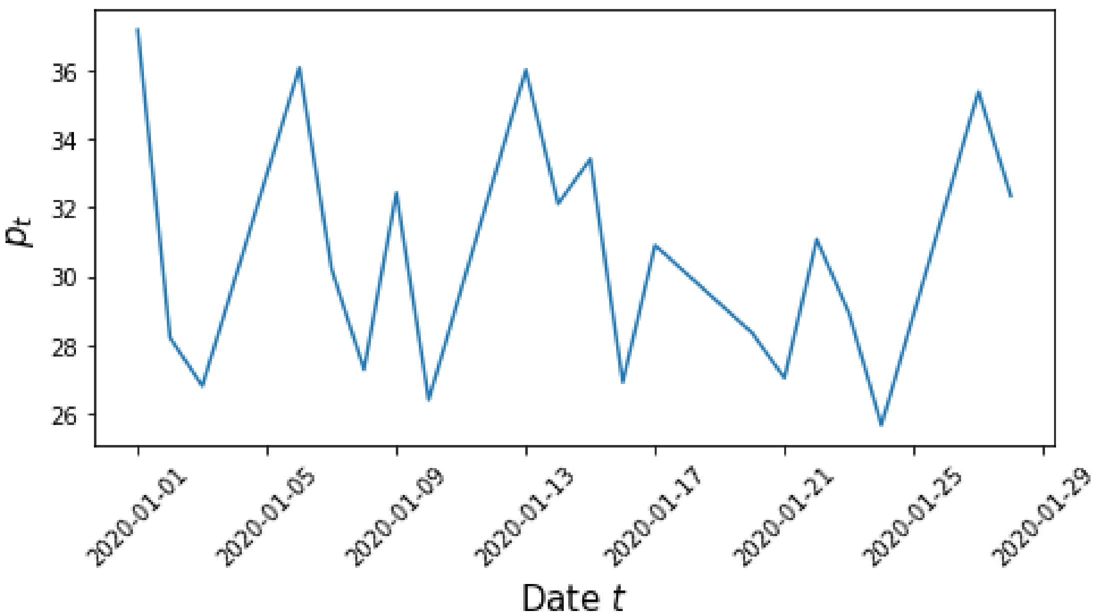
[57]:
```
fig, ax = plt.subplots(figsize=(7,4))

plt.plot(pd.date_range('2020-01-01', periods=length, freq='B'), p)

ax.set_xlabel('Date $t$', fontsize=15)
ax.set_ylabel('$p_t$', fontsize=15)

plt.xticks(rotation=45) # to rotate of 45 degrees the x-ticks

fig.tight_layout()
plt.show()
```



1.6.1 Step-by-step computation

To compute the returns, we need to calculate: - p_t as the Series `p_t` of values `p[1:]` from the second to the end; - p_{t-1} as the Series `p_tm1` of values `p[:-1]` from the beginning to the second-last one.

To be consistent with the following returns calculation, we define the two `p_t` and `p_tm1` time-series as aligned time-series indexed by business-day date range from Jan 2nd 2020 (that is, with one day of lag w.r.t. the original timeSeries).

[58]:
```
p_t = pd.Series(data=p[1:],
                index=pd.date_range('2020-01-02', periods=length-1, freq='B'),
                name="p(t)")
```

```
p_t
```

[58]:
```
2020-01-02    28.208823
2020-01-03    26.799693
2020-01-06    36.056879
2020-01-07    30.191651
2020-01-08    27.280371
2020-01-09    32.424250
2020-01-10    26.400751
2020-01-13    35.992138
2020-01-14    32.099097
2020-01-15    33.409345
2020-01-16    26.912392
2020-01-17    30.896827
2020-01-20    28.347494
2020-01-21    27.036349
2020-01-22    31.067052
2020-01-23    28.911740
2020-01-24    25.666152
2020-01-27    35.364221
2020-01-28    32.330933
Freq: B, Name: p(t), dtype: float64
```

[59]:
```
p_t.size
```

[59]:
```
19
```

[60]:
```
p_tm1 = pd.Series(data=p[:-1],
                  index=pd.date_range('2020-01-02', periods=length-1, freq='B'),
                  name="p(t-1)")
p_tm1
```

[60]:
```
2020-01-02    37.163108
2020-01-03    28.208823
2020-01-06    26.799693
2020-01-07    36.056879
2020-01-08    30.191651
2020-01-09    27.280371
2020-01-10    32.424250
2020-01-13    26.400751
2020-01-14    35.992138
2020-01-15    32.099097
2020-01-16    33.409345
2020-01-17    26.912392
2020-01-20    30.896827
2020-01-21    28.347494
2020-01-22    27.036349
```

```
2020-01-23    31.067052
2020-01-24    28.911740
2020-01-27    25.666152
2020-01-28    35.364221
Freq: B, Name: p(t-1), dtype: float64
```

[61]:
```
p_tm1.size
```

[61]:
```
19
```

We can define the linear returns

$$r_t^{\text{lin}} = p_t - p_{t-1}$$

as the `linRet_t = p_t - p_tm1` PandasSeries

[62]:
```
linRet_t = p_t - p_tm1
linRet_t.name = "linear returns r(t) = p(t) - p(t-1)"
linRet_t
```

[62]:
```
2020-01-02   -8.954285
2020-01-03   -1.409130
2020-01-06    9.257186
2020-01-07   -5.865228
2020-01-08   -2.911280
2020-01-09    5.143879
2020-01-10   -6.023500
2020-01-13    9.591387
2020-01-14   -3.893041
2020-01-15    1.310247
2020-01-16   -6.496952
2020-01-17    3.984435
2020-01-20   -2.549333
2020-01-21   -1.311146
2020-01-22    4.030703
2020-01-23   -2.155312
2020-01-24   -3.245588
2020-01-27    9.698069
2020-01-28   -3.033288
Freq: B, Name: linear returns r(t) = p(t) - p(t-1), dtype: float64
```

[63]:
```
linRet_t.size
```

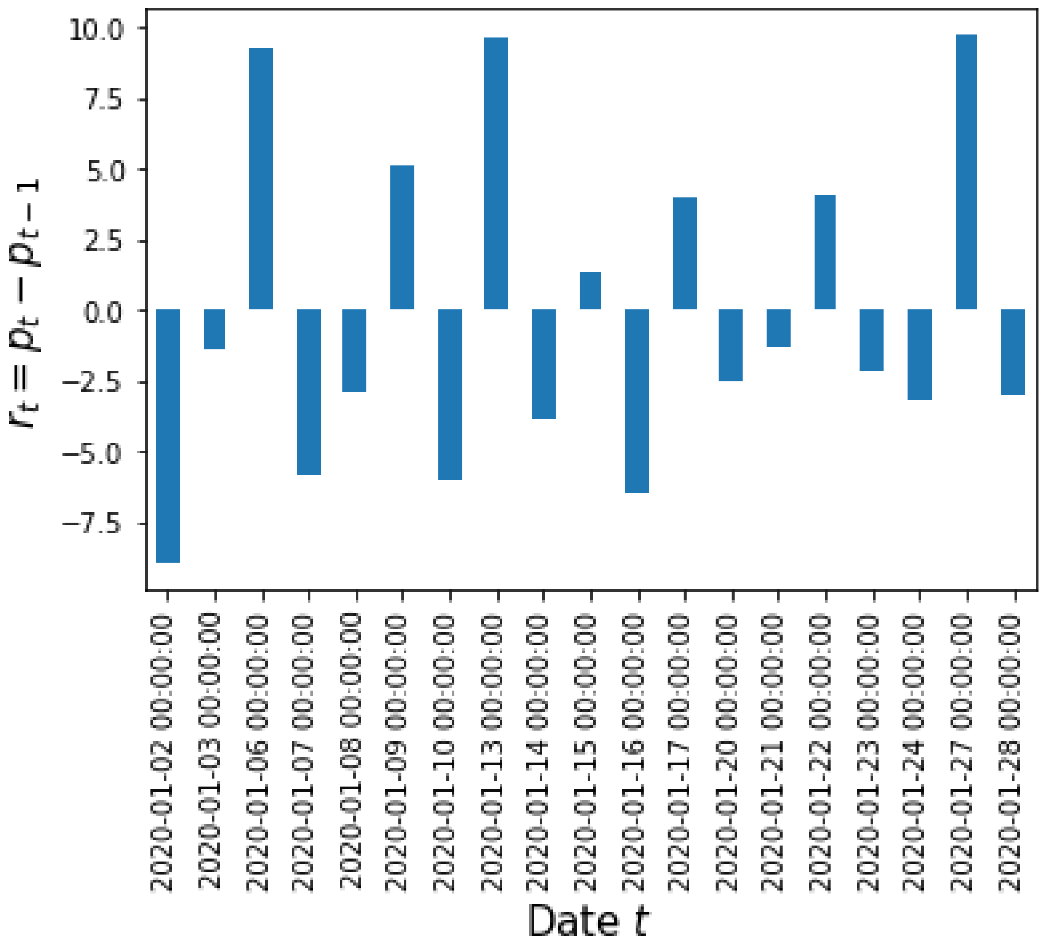
[63]:
```
19
```

and we can do a bar-plot of returns

[64]:
```python
ax = linRet_t.plot.bar()

ax.set_xlabel("Date $t$", fontsize=15)
ax.set_ylabel("$r_t = p_t - p_{t-1}$", fontsize=15)
```

[64]:
```
Text(0, 0.5, '$r_t = p_t - p_{t-1}$')
```



We can, alternatively, compute log-returns

$$r_t^{log} = \log\left(\frac{p_t}{p_{t-1}}\right)$$

as the `logRet_t = log(p_t) - log(p_tm1)` PandasSeries

[65]:
```python
logRet_t = np.log(p_t) - np.log(p_tm1)
logRet_t.name = "log-returns r(t) = log(p(t)/p(t-1))"
logRet_t
```

[65]:
```
2020-01-02   -0.275682
2020-01-03   -0.051244
2020-01-06    0.296707
2020-01-07   -0.177532
2020-01-08   -0.101398
2020-01-09    0.172739
2020-01-10   -0.205514
2020-01-13    0.309908
2020-01-14   -0.114473
2020-01-15    0.040008
2020-01-16   -0.216249
2020-01-17    0.138067
2020-01-20   -0.086115
2020-01-21   -0.047356
2020-01-22    0.138966
2020-01-23   -0.071900
2020-01-24   -0.119075
2020-01-27    0.320528
2020-01-28   -0.089676
Freq: B, Name: log-returns r(t) = log(p(t)/p(t-1)), dtype: float64
```

[66]:
```
logRet_t.size
```

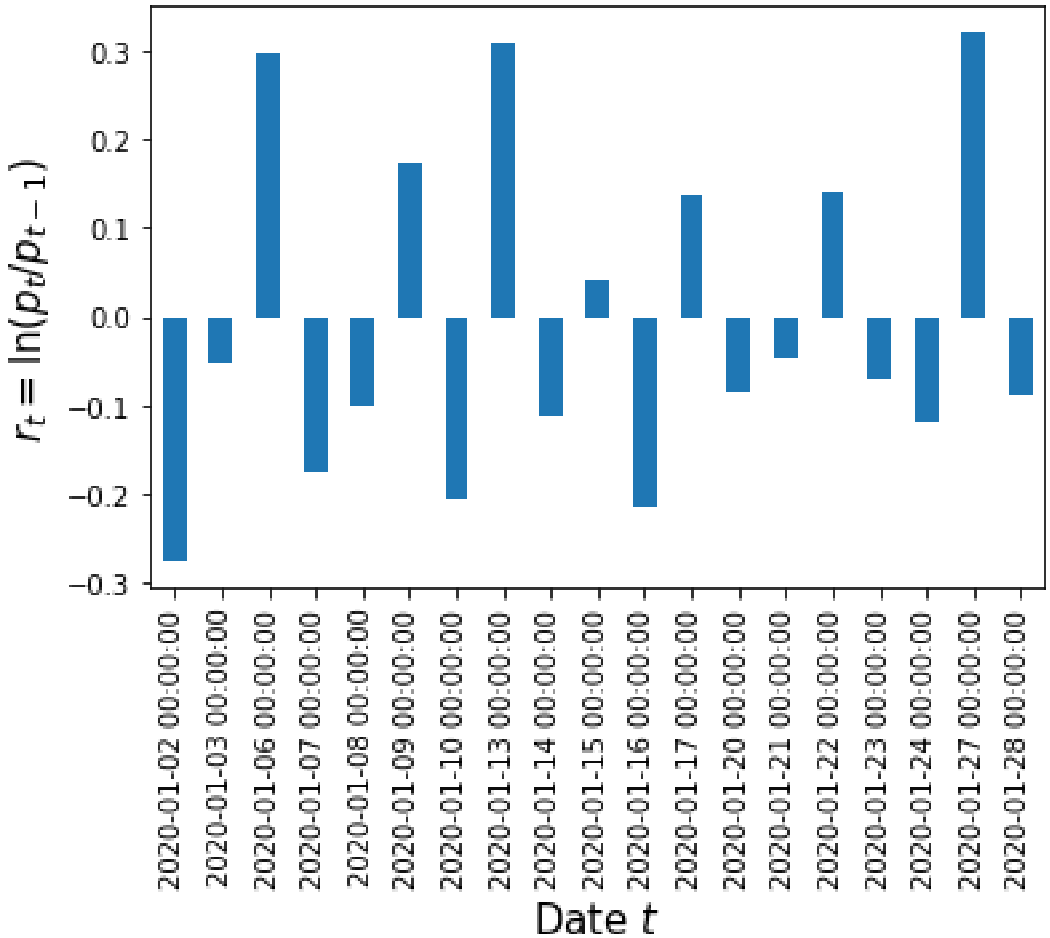
[66]:
```
19
```

and we can do a bar-plot of returns

[67]:
```
ax = logRet_t.plot.bar()

ax.set_xlabel("Date $t$", fontsize=15)
ax.set_ylabel("$r_t = \ln(p_t/p_{t-1})$", fontsize=15)
```

[67]:
```
Text(0, 0.5, '$r_t = \\ln(p_t/p_{t-1})$')
```



1.6.2 Direct computation using `.shift()`

We first (an more intuitively) define the p_t time-series as a Pandas Series out of the original (NumPy array) time-series `p`, using the whole indexing form Jan 1st 2020.

[68]:
```
p_t = pd.Series(data=p,
                index=pd.date_range('2020-01-01', periods=length, freq='B'),
                name="p(t)")
p_t
```

[68]:
```
2020-01-01    37.163108
2020-01-02    28.208823
2020-01-03    26.799693
2020-01-06    36.056879
2020-01-07    30.191651
2020-01-08    27.280371
2020-01-09    32.424250
```

```
2020-01-10    26.400751
2020-01-13    35.992138
2020-01-14    32.099097
2020-01-15    33.409345
2020-01-16    26.912392
2020-01-17    30.896827
2020-01-20    28.347494
2020-01-21    27.036349
2020-01-22    31.067052
2020-01-23    28.911740
2020-01-24    25.666152
2020-01-27    35.364221
2020-01-28    32.330933
Freq: B, Name: p(t), dtype: float64
```

Then we define the p_{t-1} time-series as the `p_tm1` Series rolling the `p_t` Series shifted of 1 business-day using the `.shift()` method. What was indexed by date t in `p_t` Series is rolled ahead of 1 bd, becoming indexed by date $t+1$ in `p_tm1`. For example, the first value 37.163108 was attributed to date 2020-01-01 in `p_t`. In `p_tm1`, 37.163108 becomes indexed by 2020-01-02.

[69]:
```
p_tm1 = p_t.shift(periods=1, freq='B')
p_tm1
```

[69]:
```
2020-01-02    37.163108
2020-01-03    28.208823
2020-01-06    26.799693
2020-01-07    36.056879
2020-01-08    30.191651
2020-01-09    27.280371
2020-01-10    32.424250
2020-01-13    26.400751
2020-01-14    35.992138
2020-01-15    32.099097
2020-01-16    33.409345
2020-01-17    26.912392
2020-01-20    30.896827
2020-01-21    28.347494
2020-01-22    27.036349
2020-01-23    31.067052
2020-01-24    28.911740
2020-01-27    25.666152
2020-01-28    35.364221
2020-01-29    32.330933
Freq: B, Name: p(t), dtype: float64
```

Notice now that the original time-series `p_t` and the rolled Series `p_tm1` are _unaligned_Series

	first index	last index	length
p_t	2020-01-01	2020-01-28	20
p_tm1	2020-01-02	2020-01-29	20

We can define the linear returns

$$r_t^{\text{lin}} = p_t - p_{t-1}$$

as the `linRet_t = p_t - p_tm1` PandasSeries.

Notice that Pandas will index `linRet_t` with the *union* of the indexes of `p_t` and `p_tm1`, filling with `NaN` the elements corresponding to indexes not shared by both (`2020-01-01` and `2020-01-29`)

[70]:
```
linRet_t = p_t - p_tm1
linRet_t.name = "linear returns r(t) = p(t) - p(t-1)"
linRet_t
```

[70]:
```
2020-01-01         NaN
2020-01-02   -8.954285
2020-01-03   -1.409130
2020-01-06    9.257186
2020-01-07   -5.865228
2020-01-08   -2.911280
2020-01-09    5.143879
2020-01-10   -6.023500
2020-01-13    9.591387
2020-01-14   -3.893041
2020-01-15    1.310247
2020-01-16   -6.496952
2020-01-17    3.984435
2020-01-20   -2.549333
2020-01-21   -1.311146
2020-01-22    4.030703
2020-01-23   -2.155312
2020-01-24   -3.245588
2020-01-27    9.698069
2020-01-28   -3.033288
2020-01-29         NaN
Freq: B, Name: linear returns r(t) = p(t) - p(t-1), dtype: float64
```

[71]:
```
linRet_t.size
```

[71]:
```
21
```

You can remove `NaN` using the `.dropna()` method

[72]:
```python
linRet_t = linRet_t.dropna()
linRet_t
```

[72]:
```
2020-01-02   -8.954285
2020-01-03   -1.409130
2020-01-06    9.257186
2020-01-07   -5.865228
2020-01-08   -2.911280
2020-01-09    5.143879
2020-01-10   -6.023500
2020-01-13    9.591387
2020-01-14   -3.893041
2020-01-15    1.310247
2020-01-16   -6.496952
2020-01-17    3.984435
2020-01-20   -2.549333
2020-01-21   -1.311146
2020-01-22    4.030703
2020-01-23   -2.155312
2020-01-24   -3.245588
2020-01-27    9.698069
2020-01-28   -3.033288
Freq: B, Name: linear returns r(t) = p(t) - p(t-1), dtype: float64
```

[73]:
```python
linRet_t.size
```

[73]:
```
19
```

Summarizing:

	first index	last index	length
`p_t`	2020-01-01	2020-01-28	20
`p_tm1`	2020-01-02	2020-01-29	20
`linRet_t = p_t - p_tm1`	2020-01-01	2020-01-29	21
`linRet_t.dropna()`	2020-01-02	2020-01-28	19

Exactly the same reasoning can be reapeated to compute the log-returns

$$r_t^{log} = \log\left(\frac{p_t}{p_{t-1}}\right)$$

defining the Series `logRet_t = log(p_t) - log(p_tm1)`, which shall have 21 rows and 2 `NaN` (first and last row) which can be dropped using `.dropna()` method. You can work it out each step by yourself.

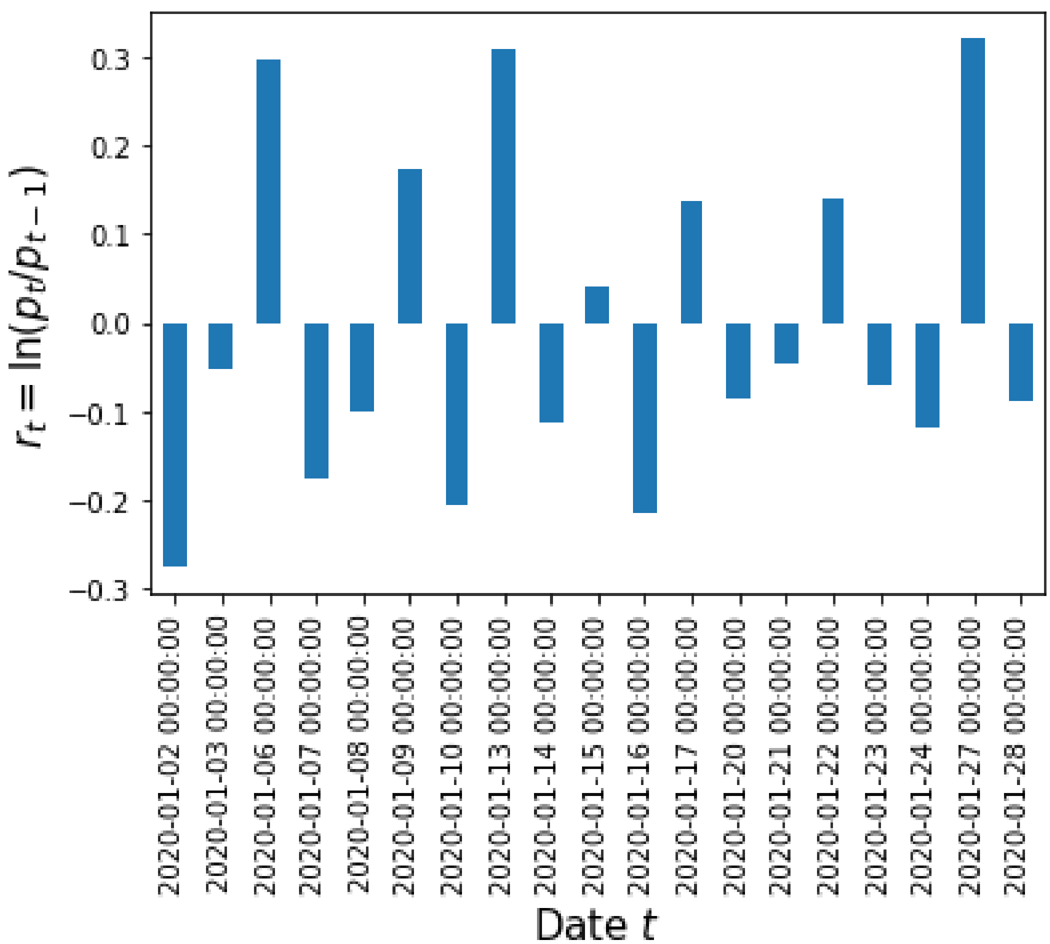
[74]:
```python
logRet_t = np.log(p_t) - np.log(p_tm1)
logRet_t.name = "log-returns r(t) = log(p(t)/p(t-1))"

logRet_t = logRet_t.dropna()

ax = logRet_t.plot.bar()

ax.set_xlabel("Date $t$", fontsize=15)
ax.set_ylabel("$r_t =  \ln(p_t/p_{t-1})$", fontsize=15)
```

[74]:
```
Text(0, 0.5, '$r_t =  \\ln(p_t/p_{t-1})$')
```



2 DataFrames

A Pandas DataFrame is a 2-dimensional labeled data structure with columns of potentially different types (Integers, Strings, Floats, etc.).

You can think of it like an Excel spreadsheet, or - knowing Pandas Series - a Dict of Series objects.

2.1 Creation: `pd.DataFrame()`

DataFrames can be created using the constructor:

`pd.DataFrame(data[, index, columns])`

where:

- `data`: is the data content of the DataFrame. It can be a NumPy N-dim array, a Python Dict of: - Pandas Series, - 1-dim NumPy arrays, - Python Lists, - etc…
- `index`: (optional) is the index of the DataFrame. It can be an array-like structure (e.g. a List). If not provided, default index spans the rows of `data`. For example, if `data` is a NumPy `(n,m)` array, it is `[0,1,...,n-1]`.
- `columns`: (optional)Lists the columns of the DataFrame. It can be an array-like strucutre (e.g. a List). If not provided, default columns spans the columns of `data`. For example, if `data` is a NumPy `(n,m)` array, it is `[0,1,...,m-1]`.

Here we consider the creation of a Pandas DataFrame from a 2-dimensional NumPy array (that is, a matrix). We refer to Intro to data structures - DataFrame for other creational paradigms and full details.

So, let's define a 10×5 matrix with columns the number from 1 to 10 raised to powers 1, 2, 3, 4, 5

[75]:
```python
mat = np.array([[i**k for i in range(1,11)] for k in range(1,6)]).T
mat
```

[75]:
```
array([[     1,      1,      1,      1,      1],
       [     2,      4,      8,     16,     32],
       [     3,      9,     27,     81,    243],
       [     4,     16,     64,    256,   1024],
       [     5,     25,    125,    625,   3125],
       [     6,     36,    216,   1296,   7776],
       [     7,     49,    343,   2401,  16807],
       [     8,     64,    512,   4096,  32768],
       [     9,     81,    729,   6561,  59049],
       [    10,    100,   1000,  10000, 100000]])
```

[76]:
```python
mat.shape
```

[76]:
```
(10, 5)
```

we pass the matrix `mat` as the `data` parameter of `pd.DataFrame()`

[77]:
```python
df = pd.DataFrame(mat)
df
```

[77]:

	0	1	2	3	4
0	1	1	1	1	1

1	2	4	8	16	32
2	3	9	27	81	243
3	4	16	64	256	1024
4	5	25	125	625	3125
5	6	36	216	1296	7776
6	7	49	343	2401	16807
7	8	64	512	4096	32768
8	9	81	729	6561	59049
9	10	100	1000	10000	100000

The function `pd.DataFrame()` returns a Pandas DataFrame object. Each element in the table is linked to its corresponding index and column.

Notice that: - the `index` which is generated by default (since we didn't provide one explicitly) is `0, 1,..., 9 = mat.shape[0]-1`; - the `columns` which are generated by default (since we didn't provide them explicitly) are `0, 1,..., 4 = mat.shape[1]-1`;

```
[78]: type(df)
```

```
[78]: pandas.core.frame.DataFrame
```

Notice that the explicit assignment `data=mat` is optional and equivalent to `pd.DataFrame(mat)`.

Similarly to NumPy arrays and Pandas Series, Pandas DataFrames have meta-informative attributes too. Let's have a look at some of them.

The number of elements is given by

```
[79]: df.size
```

```
[79]: 50
```

The number of rows and columns of the DataFrame is, similarly to NumPy arrays:

```
[80]: df.shape
```

```
[80]: (10, 5)
```

Each column of the DataFrame may have different data-type, use `.dtypes` attribute to retrieve them (mind the plural)

```
[81]: df.dtypes
```

```
[81]: 0    int32
      1    int32
      2    int32
      3    int32
      4    int32
      dtype: object
```

Notice that a `pd.Series` is returned with each column's data-type reported as a `str` (`object` is the Pandas for `str` data-type) and linked to an index labelled as the corresponding column label in the DataFrame (another example later).

Similarly, to Series, you can directly access the index sequence:

[82]:
```
df.index
```

[82]:
```
RangeIndex(start=0, stop=10, step=1)
```

As we have seen for Series, `RangeIndex` is the kind of `[0,1,...,mat.shape[0]-1]` index which Pandas creates by default when you don't input one explicitly.

[83]:
```
df.columns
```

[83]:
```
RangeIndex(start=0, stop=5, step=1)
```

Similarly, a `[0,1,...,mat.shape[1]-1]` `RangeIndex` is created to label the columns when you don't provide them explicitly.

Of course we can give more descriptive names to the columns of our DataFrame:

[84]:
```
df.columns = ['x', 'x^2', 'x^3', 'x^4', 'x^5']
df
```

[84]:
```
    x  x^2   x^3    x^4     x^5
0   1    1     1      1       1
1   2    4     8     16      32
2   3    9    27     81     243
3   4   16    64    256    1024
4   5   25   125    625    3125
5   6   36   216   1296    7776
6   7   49   343   2401   16807
7   8   64   512   4096   32768
8   9   81   729   6561   59049
9  10  100  1000  10000  100000
```

[85]:
```
df.columns
```

[85]:
```
Index(['x', 'x^2', 'x^3', 'x^4', 'x^5'], dtype='object')
```

Now the columns that we define are a general `Index` of Strings (`dtype='object'`).

As for Pandas Series, ff you want just the values (without the indexing) - that is, the original NumPy `mat` in our case - these can be accessed through the `.values` attribute

[86]:
```
df.values
```

[86]:
```
array([[     1,      1,      1,      1,      1],
       [     2,      4,      8,     16,     32],
```

```
 [     3,      9,     27,     81,    243],
 [     4,     16,     64,    256,   1024],
 [     5,     25,    125,    625,   3125],
 [     6,     36,    216,   1296,   7776],
 [     7,     49,    343,   2401,  16807],
 [     8,     64,    512,   4096,  32768],
 [     9,     81,    729,   6561,  59049],
 [    10,    100,   1000,  10000, 100000]])
```

2.1.1 Time indexes: pd.date_range() and pd.to_datetime()

In exactly the same way we were able to define time-indexes for Pandas Series, we can do it for Pandas DataFrames.

Here we create a range of business days (denoted by the *frequency* `freq='B'`) starting from Jan 1st 2020. The range lasts a number of `periods` equal to the rows of our DataFrame (`df.shape[0]` which would of course be the same as `mat.shape[0]`)

[87]:
```
dates = pd.date_range('2020-01-01', periods=df.shape[0], freq='B')
dates
```

[87]:
```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
               '2020-01-13', '2020-01-14'],
              dtype='datetime64[ns]', freq='B')
```

The kind of time-index that is returned is a `DatetimeIndex`. We can re-index our DataFrame with this new index.

[88]:
```
df.index = dates
df
```

[88]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[89]:
```
df.index
```

[89]:
```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
```

```
               '2020-01-13', '2020-01-14'],
              dtype='datetime64[ns]', freq='B')
```

Of course, we could have defined the DataFrame with our desired `data`, `index` and `columns` setup.

[90]:
```
df = pd.DataFrame(data=mat,
                  index=dates,
                  columns=['x', 'x^2', 'x^3', 'x^4', 'x^5'])
df
```

[90]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Function `pd.to_datetime()` has already been introduced in dedicated Data Analysis - Introduction to Pandas Series notebook as a converter from a List of dates Strings to a `DatetimeIndex` object.

[91]:
```
pd.to_datetime(['2020-01-02', '2020-01-07', '2020-01-10'])
```

[91]:
```
DatetimeIndex(['2020-01-02', '2020-01-07', '2020-01-10'],
dtype='datetime64[ns]', freq=None)
```

As well as for Pandas Series, this function allows to filter rows of a DataFrame according to a List of Strings representing dates. We'll see it shortly.

2.2 Basic plotting: `df.plot()` and `df.plot.bar()`

As with Pandas Series, plotting a DataFrame cannot be easier

[92]:
```
df
```

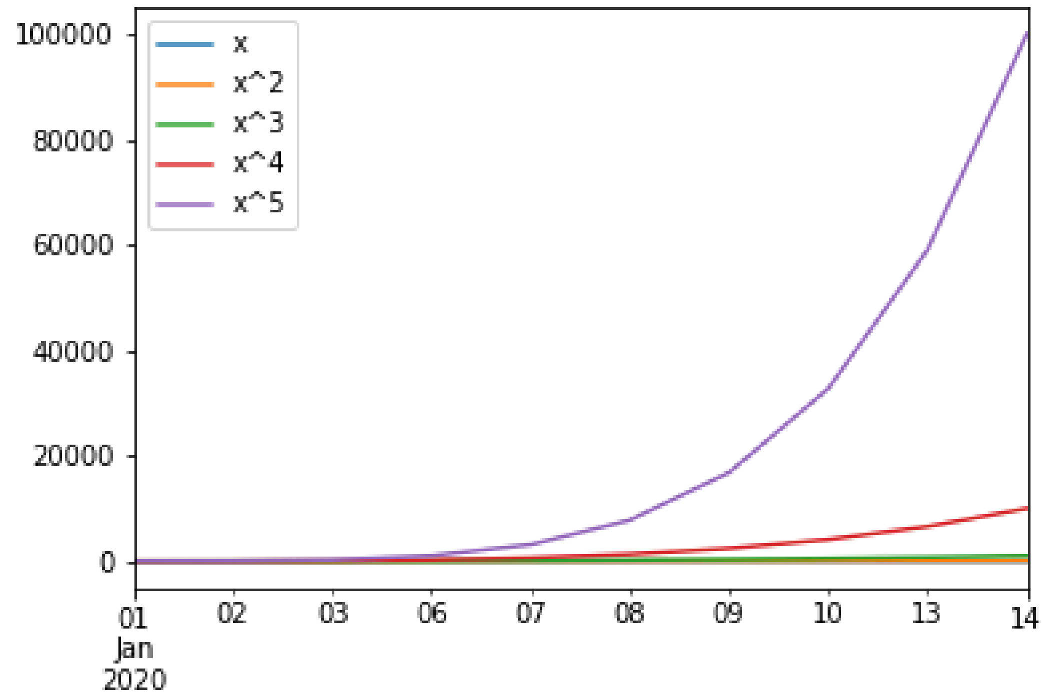
[92]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768

2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[93]:
```
df.plot()
```

[93]: <matplotlib.axes._subplots.AxesSubplot at 0x190855d5d48>



Notice that each columns, as reported in the picture legend, is translated into a line of different color, wherheas the common index is used on the x-axis to draw syncronized values.

Ok, given that our data are power functions, which grow at different speeds, it's difficult to appreciate all them together in the same plot… if we plot them straight-away. Let's plot their logs!

So let's define a new DataFrame `df_log` which has the same index and columns of the original `df`, but each element get's transformed throug a ln() function.

We anticipate here the flexibility of NumPy's universal functions, which most can be used with `pandas.DataFrame` parameters in input (instead of `numpy.ndarray`).

En passant, we re-label the columns of the `df_log` too. Notice the use ofList comprehension together with the `+` operator to concatenate `str`.

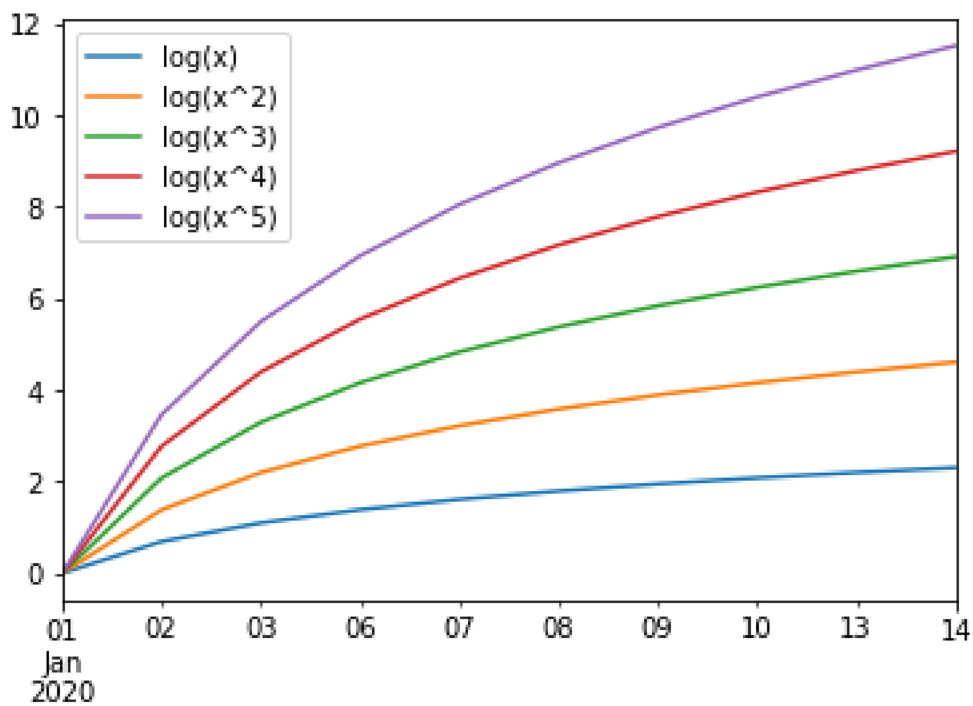
[94]:
```
df_log = np.log(df)
df_log.columns = ['log(' + c + ')' for c in df.columns]
df_log
```

[94]:

	log(x)	log(x^2)	log(x^3)	log(x^4)	log(x^5)
2020-01-01	0.000000	0.000000	0.000000	0.000000	0.000000
2020-01-02	0.693147	1.386294	2.079442	2.772589	3.465736
2020-01-03	1.098612	2.197225	3.295837	4.394449	5.493061
2020-01-06	1.386294	2.772589	4.158883	5.545177	6.931472
2020-01-07	1.609438	3.218876	4.828314	6.437752	8.047190
2020-01-08	1.791759	3.583519	5.375278	7.167038	8.958797
2020-01-09	1.945910	3.891820	5.837730	7.783641	9.729551
2020-01-10	2.079442	4.158883	6.238325	8.317766	10.397208
2020-01-13	2.197225	4.394449	6.591674	8.788898	10.986123
2020-01-14	2.302585	4.605170	6.907755	9.210340	11.512925

[95]:
```
df_log.plot()
```

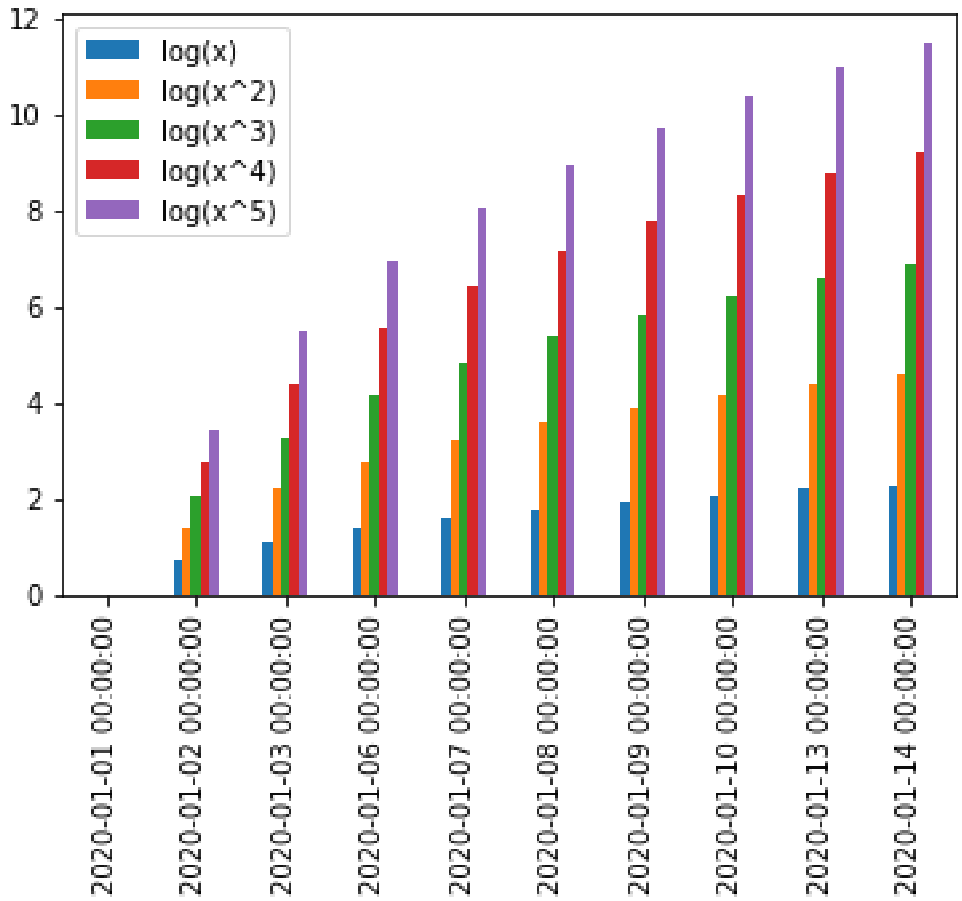
[95]: <matplotlib.axes._subplots.AxesSubplot at 0x19085652c48>



Now everything looks much clearer. As we did for Pandas Series, of course we can produce bar-plots (as well as the tons of visualization styles which are available in Pandas

[96]:
```
df_log.plot.bar()
```

[96]: <matplotlib.axes._subplots.AxesSubplot at 0x190856daa88>



2.3 Indexing and Slicing

In this section we describe the different possibilities that you have to access elements of a DataFrame `df`. You can:

- Select columns using the `[]` access operator;
- Filter rows according to a logical condition using the `[]` access operator;
- Select specific rows and columns using column names using `.loc[]` access operator;
- Select specific rows and columns using numerical positional indexes in the table using `.iloc[]` access operator.

We consider each case in a separate section with examples using our DataFrame of power functions.

[97]:
```
df
```

[97]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024

```
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

2.3.1 Selecting columns: []

In this section we describe how you can select entire columns of a DataFrame. You can use the [] access operator. Here is an overview:

	syntax	example
select 1 column	`df[colName]`	`df['x^2']`
select ≥ 2 columns	`df[[listOfCols]]`	`df[['x^2', 'x^3', 'x^5']]`

We can select a column using the column name inside the [] access operator, as if the DataFrame was a Dict of the column-Series with column names its keys

[98]:
```
s_x2 = df['x^2']
s_x2
```

[98]:
```
2020-01-01      1
2020-01-02      4
2020-01-03      9
2020-01-06     16
2020-01-07     25
2020-01-08     36
2020-01-09     49
2020-01-10     64
2020-01-13     81
2020-01-14    100
Freq: B, Name: x^2, dtype: int32
```

Notice that - having selected just one column - a Pandas Series is returned

[99]:
```
type(s_x2)
```

[99]:
```
pandas.core.series.Series
```

If you want to select more than one column, just wrap their names in a List

[100]:
```
df_x235 = df[['x^2', 'x^3', 'x^5']]
df_x235
```

[100]:
```
            x^2  x^3    x^5
2020-01-01    1    1      1
```

```
2020-01-02    4     8      32
2020-01-03    9    27     243
2020-01-06   16    64    1024
2020-01-07   25   125    3125
2020-01-08   36   216    7776
2020-01-09   49   343   16807
2020-01-10   64   512   32768
2020-01-13   81   729   59049
2020-01-14  100  1000  100000
```

and notice that a new DataFrame is returned

[101]:
```
type(df_x235)
```

[101]:
```
pandas.core.frame.DataFrame
```

2.3.2 Conditional Selection: filtering rows

In this section we describe how you can filter specific rows according to a logical condition. You can input a logical condition `logicalCondition` to the `[]` access operator, like `df[logicalCondition]`. Here is an overview:

examples	meaning
`df[df['x^2'] > 5]`	all rows s.t. values in 'x^2' col are > 5
`df[df['x^2'] <= df['x']]`	all rows s.t. values on 'x^2' col are <= than values in 'x' col
`df[df['rating'].isin(['AAA', 'AA', 'A'])]`	all rows s.t. values in 'rating' col are in theList ['AAA', 'AA', 'A']

[102]:
```
df
```

[102]:
```
             x  x^2   x^3    x^4     x^5
2020-01-01   1    1     1      1       1
2020-01-02   2    4     8     16      32
2020-01-03   3    9    27     81     243
2020-01-06   4   16    64    256    1024
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

Suppose that you want to filter the rows for which column `x^2` is greater than a given threshold, say 17. The logical condition to achieve this is

[103]:
```
df['x^2'] > 17
```

```
[103]: 2020-01-01    False
2020-01-02    False
2020-01-03    False
2020-01-06    False
2020-01-07     True
2020-01-08     True
2020-01-09     True
2020-01-10     True
2020-01-13     True
2020-01-14     True
Freq: B, Name: x^2, dtype: bool
```

Notice that what is returned is a Pandas Series that is named after column `x^2` and that has `True`/`False` boolean values according to whether the condition is satisfied or not.

You can use this Series of boolean values to actually filter rows of the original DataFrame

```
[104]: df[df['x^2'] > 17]
```

[104]:

	x	x^2	x^3	x^4	x^5
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Notice that the output is the original DataFrame (all columns) with only the rows for that the Series `df['x^2'] > 17` has `True` value. That is to say, only those rows for which the conditon $x^2 > 17$ is satisfied.

The logical condition might involve other rows too. For example, let's define a new DataFrame with power-like columns of decimal number $d \in [0, 2]$ range

```
[105]: mat_decimal = np.array([[i**k for i in np.linspace(0,2,10)] for k in␣
       ↪range(1,6)]).T
```

```
[106]: x_axis = [i for i in np.linspace(0,2,10)]
```

```
[107]: df_decimal = pd.DataFrame(data = mat_decimal,
                                 index = x_axis,
                                 columns = df.columns)

       df_decimal
```

[107]:

	x	x^2	x^3	x^4	x^5
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.222222	0.222222	0.049383	0.010974	0.002439	0.000542
0.444444	0.444444	0.197531	0.087791	0.039018	0.017342

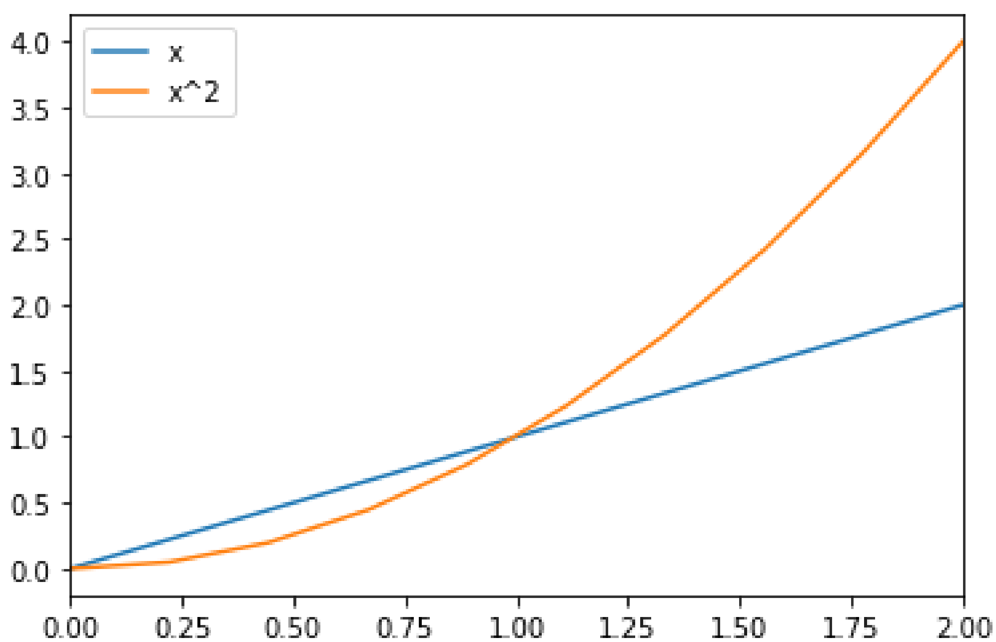
0.666667	0.666667	0.444444	0.296296	0.197531	0.131687
0.888889	0.888889	0.790123	0.702332	0.624295	0.554929
1.111111	1.111111	1.234568	1.371742	1.524158	1.693509
1.333333	1.333333	1.777778	2.370370	3.160494	4.213992
1.555556	1.555556	2.419753	3.764060	5.855205	9.108097
1.777778	1.777778	3.160494	5.618656	9.988721	17.757727
2.000000	2.000000	4.000000	8.000000	16.000000	32.000000

Notice that to index the DataFrame we use a standard grid of the $[0, 2]$ segment on the x-axis, made of 10 linearly spaced points. Of course, it coincides with column `x`, as the latter represents the identity function $f(x) = x$.

Let's focus on the first two columns `x` and `x^2` and let's see for which values of the x-axis the $x^2 \leq x$. A plot might helps here:

[108]:
```
df_decimal[['x', 'x^2']].plot()
```

[108]: `<matplotlib.axes._subplots.AxesSubplot at 0x190857f4508>`



Do you see it? The touching point $x = x^2$ is at $x = 1$. For $x < 1$, $x^2 < x$. Let's filter rows according to this condition

[109]:
```
df_decimal['x^2'] <= df_decimal['x']
```

[109]:
```
0.000000     True
0.222222     True
0.444444     True
```

```
0.666667     True
0.888889     True
1.111111    False
1.333333    False
1.555556    False
1.777778    False
2.000000    False
dtype: bool
```

again, we can use this Series of booleans to filter rows of `df_decimal`

[110]:
```python
df_decimal[df_decimal['x^2'] <= df_decimal['x']]
```

[110]:
```
                 x       x^2       x^3       x^4       x^5
0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
0.222222  0.222222  0.049383  0.010974  0.002439  0.000542
0.444444  0.444444  0.197531  0.087791  0.039018  0.017342
0.666667  0.666667  0.444444  0.296296  0.197531  0.131687
0.888889  0.888889  0.790123  0.702332  0.624295  0.554929
```

which, as expected, returns only rows for which the `x_axis` index is at most 1.

Finally, you may want to filter rows according to values in a List. Suppose we have a List of Standard & Poor's *high grade* credit ratings

[111]:
```python
highGradeRatings = ['AA+', 'AA', 'AA-']
```

and a DataFrame of some reference data (country, S&P credit ratings and corresponding spreads in bps)

[112]:
```python
df_refData = pd.DataFrame(data={
                              'S&P Rating': ['A', 'BB', 'AA', 'CCC'],
                              'Spread': [100, 300, 70, 700],
                              'Country': ['USA', 'ITA', 'UK', 'ITA']
                             },
                        index=['Firm_1', 'Firm_2', 'Firm_3', 'Firm_4'])

df_refData
```

[112]:
```
       S&P Rating  Spread Country
Firm_1          A     100     USA
Firm_2         BB     300     ITA
Firm_3         AA      70      UK
Firm_4        CCC     700     ITA
```

Notice how we have used a `Dict` of Python `list` as `data` parameter in `pd.DataFrame` constructor, which provides both values and column names (which allows to skip `columns` parameter).

So now suppose we want only want the rows (the firms) having a credit rating which is among the `highGradeRatings`.

You can do this row-filtering using the `.isin()` method, which tests whether values of a column are in a List

[113]:
```
df_refData['S&P Rating'].isin(highGradeRatings)
```

[113]:
```
Firm_1    False
Firm_2    False
Firm_3     True
Firm_4    False
Name: S&P Rating, dtype: bool
```

and returns a Pandas Series of bools. You can use this boolean Series to filter rows

[114]:
```
df_refData[df_refData['S&P Rating'].isin(highGradeRatings)]
```

[114]:

	S&P Rating	Spread	Country
Firm_3	AA	70	UK

which, as expected, returns only row for `Firm_3`, that is the only firm with a rating in the `highGradeRatings`List.

2.3.3 Selecting rows with rows and columns *names* : `.loc[]`

In this section we describe how you can select specific rows and columns using their labels. You can use the `.loc[ row indexer, column indexer ]` access operator using labels to identify rows and columns. For both the `row indexer`, before the comma, and the `column indexer`, after the comma, you can have several options. We review them in the next sub-sections.

[115]:
```
df
```

[115]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Of course you can combine different kind of indexers (single,List, slice, conditonal) for row and col indexers separately.

2.3.3.1. Single label A single label can be used to get one row and/or column only:

row indexer	col indexer	syntax/example	meaning
label	label	`df.loc[indexLabel, colName]`	element in col `colName` at row `indexLabel`
	`df.loc['2020-01-02', 'x^2']`		

Let’s select the single element in column `x^2` and at row of index ‘2020-01-06’

[116]:
```python
df.loc['2020-01-06', 'x^2']
```

[116]:
```
16
```

Typically, accessing using an index and/or a column that doesn’t exist, raises a `KeyError`

[117]:
```python
# raises KeyError as '2020-02-01' is not an index of df
#
# df.loc['2020-02-01', 'x^2']
```

[118]:
```python
# raises KeyError as 'x^7' is not a column of df
#
# df.loc['2020-01-06', 'x^7']
```

2.3.3.2. List of labels a List of labels can be used to get a List of rows and/or columns, using `pd.to_datetime()` in case of (typically) indexes which are Dates Strings:

row indexer	col indexer	syntax/example	meaning
…	namesList	`df.loc[..., [listOfCols]]`	elements in `[listOfCols]` cols at rows…
	`df.loc[..., ['x^2', 'x^3', 'x^5']]`		
labelsList	…	`df.loc[[listOfIndexLabels], ...]`	elements at `[listOfIndexLabels]` rows and in cols…
	`df.loc[['a', 'c', 'd'], ...]`		
datesList	…	`df.loc[pd.to_datetime([listOfDatesStrings]), ...]`	elements at `[listOfDatesStrings]` rows and in cols…
	`df.loc[pd.to_datetime(['2020-01-02', '2020-01-05']), ...]`		

Let's define a List of columns and rows to use for the selection. Let's use `df_refData` for this example

[119]:
```
df_refData
```

[119]:
```
       S&P Rating  Spread Country
Firm_1          A     100     USA
Firm_2         BB     300     ITA
Firm_3         AA      70      UK
Firm_4        CCC     700     ITA
```

[120]:
```
referenceData = ['S&P Rating', 'Spread']
firms = ['Firm_1', 'Firm_4']
```

and select rows and/or columns accordingly

[121]:
```
df_refData.loc[firms, referenceData]
```

[121]:
```
       S&P Rating  Spread
Firm_1          A     100
Firm_4        CCC     700
```

Let's go back to `df`. DataFrame `df` has Dates as index (a `DatetimeIndex`)

[122]:
```
df.index
```

[122]:
```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
               '2020-01-13', '2020-01-14'],
              dtype='datetime64[ns]', freq='B')
```

as already noticed for Pandas Series, when you want to filter rows according to a List of Strings representing dates, you have to convert them into a valid `DatetimeIndex` object using `pd.to_datetime`

So let's choose a few dates and cols

[123]:
```
someDates = ['2020-01-03', '2020-01-06', '2020-01-09']
oddsPowerCols = ['x', 'x^3', 'x^5']
```

[124]:
```
df.loc[pd.to_datetime(someDates), oddsPowerCols]
```

[124]:
```
            x  x^3    x^5
2020-01-03  3   27    243
2020-01-06  4   64   1024
2020-01-09  7  343  16807
```

as expected, the desired selection of odds powers at the requested dates is returned.

As already noticed for Pandas Series (*repetita iuvant*) using the dates Strings straight away as row indexer would raise a `KeyError` because Pandas tries to look for those raw stirngs among the indexes, whereas what you actually meant were dates…

[125]:
```python
# raises KeyError
#
# df.loc[someDates, oddsPowerCols]
```

2.3.3.3. Slice of labels A slice of labels can be used to get a slice of rows and/or columns:

row indexer	col indexer	syntax/example	meaning
slice	…	`df.loc[fromIndexLabel:toIndexLabel:eachTotRows, ...]`	elements at rows as per slice and in cols…
	`df.loc['2020-01-06':'2020-01-09', ...]`		
…	slice	`df.loc[..., fromColName:toColName:eachTotCols]`	elements in cols as per slice and at rows…
	`df.loc[..., 'x':'x^3']`		

For example, we may be interested to the last few dates values of odds powers columns

[126]:
```python
df.loc['2020-01-09':, oddsPowerCols]
```

[126]:

	x	x^3	x^5
2020-01-09	7	343	16807
2020-01-10	8	512	32768
2020-01-13	9	729	59049
2020-01-14	10	1000	100000

where the slice `'2020-01-09':` selects all the dates indexes from '2020-01-09' to the end, whereas we use a List of column names as cols indexer.

You may instead be interested in a slice of columns. This is possible too…

[127]:
```python
df.loc['2020-01-09', 'x':'x^3']
```

[127]:
```
x        7
x^2     49
x^3    343
Name: 2020-01-09 00:00:00, dtype: int32
```

Notice that a Series is returned.

Of course, you can use simultaneously slices both for row indexers and col indexers

[128]:
```python
df.loc['2020-01-09':, 'x':'x^3']
```

```
[128]:              x  x^2   x^3
2020-01-09   7   49   343
2020-01-10   8   64   512
2020-01-13   9   81   729
2020-01-14  10  100  1000
```

that, as expected returns a DataFrame.

2.3.3.4. Conditional Expression A logical conditional expression can be used to filter rows and/or columns (typically according to their names):

row indexer	col indexer	syntax/example	meaning
logical condition	…	`df.loc[logicalCondition, ...]`	elements at rows filtered as per logical condition and in cols…
	`df.loc[df['x^2'] > 5, ...]`		
…	logical condition	`df.loc[..., df.columns.isin(listOfNames)]`	elements in cols in `listOfNames` and at rows…
	`df.loc[..., df.columns.isin(['Pippo', 'x^3', 'Pluto'])]`		

We can use `df_decimal` to make examples

```
[129]: df_decimal
```

```
[129]:              x       x^2       x^3        x^4        x^5
0.000000  0.000000  0.000000  0.000000   0.000000   0.000000
0.222222  0.222222  0.049383  0.010974   0.002439   0.000542
0.444444  0.444444  0.197531  0.087791   0.039018   0.017342
0.666667  0.666667  0.444444  0.296296   0.197531   0.131687
0.888889  0.888889  0.790123  0.702332   0.624295   0.554929
1.111111  1.111111  1.234568  1.371742   1.524158   1.693509
1.333333  1.333333  1.777778  2.370370   3.160494   4.213992
1.555556  1.555556  2.419753  3.764060   5.855205   9.108097
1.777778  1.777778  3.160494  5.618656   9.988721  17.757727
2.000000  2.000000  4.000000  8.000000  16.000000  32.000000
```

So, let's go back to our $x^2 \leq x$ logical condition

```
[130]: df_decimal['x^2'] <= df_decimal['x']
```

[130]:
```
0.000000     True
0.222222     True
0.444444     True
0.666667     True
0.888889     True
1.111111    False
1.333333    False
1.555556    False
1.777778    False
2.000000    False
dtype: bool
```

Which you can use as a row indexer. If you are interested in all the columns, then fine, we have already seen that you can just use the [] access operator with the logical condition only (row filtering)

[131]:
```
df_decimal[df_decimal['x^2'] <= df_decimal['x']]
```

[131]:

	x	x^2	x^3	x^4	x^5
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.222222	0.222222	0.049383	0.010974	0.002439	0.000542
0.444444	0.444444	0.197531	0.087791	0.039018	0.017342
0.666667	0.666667	0.444444	0.296296	0.197531	0.131687
0.888889	0.888889	0.790123	0.702332	0.624295	0.554929

Nevertheless, if you are not interested in all the columns, but say `x` and `x^2`, you may wrap them in a List to get only them back

[132]:
```
df_decimal.loc[df_decimal['x^2'] <= df_decimal['x'], ['x', 'x^2']]
```

[132]:

	x	x^2
0.000000	0.000000	0.000000
0.222222	0.222222	0.049383
0.444444	0.444444	0.197531
0.666667	0.666667	0.444444
0.888889	0.888889	0.790123

and notice that we need the `.loc[]` access operator since we are using column indexers too.

Let's go back to `df`

[133]:
```
df.columns
```

[133]:
```
Index(['x', 'x^2', 'x^3', 'x^4', 'x^5'], dtype='object')
```

Suppose that you have a List o some names

[134]:
```
listOfNames = ['x', 'x^x', 'Pluto', 'x^3', 'y']
```

as we see, some of them are columns of `df`, but there are others which are otherwise unrelated names.

Well, you can use the `.isin()` method of the `df.columns` attribute, which returns the subset of `df` columns which are in `listOfNames`, and can be used as a column indexer

[135]:
```
df.loc['2020-01-09':, df.columns.isin(listOfNames)]
```

[135]:

	x	x^3
2020-01-09	7	343
2020-01-10	8	512
2020-01-13	9	729
2020-01-14	10	1000

which, as expected, returns all the element at rows from date '2020-01-09' (slice of labels) but only for the `x` and `x^3` columns, which are the only two in the `listOfNames`.

Notice that, if no column of `df.columns` is in `listOfNames`, a DataFrame without any column is returned

[136]:
```
unrelatedListOfNames = [ 'Donald Duck', 'Mickey Mouse']
```

[137]:
```
df.loc['2020-01-09':, df.columns.isin(unrelatedListOfNames)]
```

[137]:
```
Empty DataFrame
Columns: []
Index: [2020-01-09 00:00:00, 2020-01-10 00:00:00, 2020-01-13 00:00:00,
2020-01-14 00:00:00]
```

quite a strange object, I agree. Just remember this is caused by the fact that no column of `df` was in `unrelatedLisOfNames`.

2.3.3.5. Colon : A colon : can be used to specify that you want to select all rows or columns:

row indexer	col indexer	syntax/example	meaning
:	:	`df.loc[:, :]`	all the elements (equivalent to just `df`)
:	…	`df.loc[:, ...]`	elements from all the rows and in cols…
…	:	`df.loc[..., :]`	elements from all the cols and at rows…
…	omitted	`df.loc[...]`	equivalent to `df.loc[..., :]`

When you want the entire DataFrame you are used to just type `df`

[138]:
```
df
```

[138]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243

```
2020-01-06   4   16    64    256    1024
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

and keep doing it. Just rememember that writing `df` is actually interpreted as a selection of all rows and columns of the DataFrame, omitting rows and cols indexers. An equivalent way to select all rows and cols, without omitting the indexers is `df.loc[:,:]`

[139]:
```
df.loc[:,:]
```

[139]:
```
             x  x^2   x^3    x^4     x^5
2020-01-01   1    1     1      1       1
2020-01-02   2    4     8     16      32
2020-01-03   3    9    27     81     243
2020-01-06   4   16    64    256    1024
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

If you want all rows but just some columns you can use the colon `:` as a row indexer

[140]:
```
df.loc[:, ['x', 'x^2']]
```

[140]:
```
             x  x^2
2020-01-01   1    1
2020-01-02   2    4
2020-01-03   3    9
2020-01-06   4   16
2020-01-07   5   25
2020-01-08   6   36
2020-01-09   7   49
2020-01-10   8   64
2020-01-13   9   81
2020-01-14  10  100
```

In the same way, if you want all the columns, but just a few rows, you can use the colon `:` as a cols indexer

[141]:
```
df.loc['2020-01-09':, :]
```

[141]:

	x	x^2	x^3	x^4	x^5
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Notice that if you want all the columns, you can completely omit the column indexer. Therefore, `df.loc['2020-01-09':, :]` is equivalent to `df.loc['2020-01-09':]`

[142]:
```
df.loc['2020-01-09':]
```

[142]:

	x	x^2	x^3	x^4	x^5
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Finally notice, that the omission of indexers doens't work symmetrically. You cannot omit the row indexer.

That is to say, if you want all the rows but just the `['x', 'x^2']` cols, you can do

[143]:
```
df.loc[:, ['x', 'x^2']]
```

[143]:

	x	x^2
2020-01-01	1	1
2020-01-02	2	4
2020-01-03	3	9
2020-01-06	4	16
2020-01-07	5	25
2020-01-08	6	36
2020-01-09	7	49
2020-01-10	8	64
2020-01-13	9	81
2020-01-14	10	100

but you cannot do

[144]:
```
# raises KeyError
#
# df.loc[['x', 'x^2']]
```

When using the `.loc[]` access specifier omitting an indexer, the missing indexer is interpreted as a columns indexer, that's why something like

```
df.loc[['x', 'x^2']]
```

is interpreted as

```
df.loc[['x', 'x^2'], :]
```

which makes Pandas looking for rows with names `x` and `x^2` and thus raising a `KeyError` not finding them.

By the way, you already know how to select all the rows and few cols: you have to use the `[]` access operator straight away

[145]:
```
df[['x', 'x^2']]
```

[145]:

	x	x^2
2020-01-01	1	1
2020-01-02	2	4
2020-01-03	3	9
2020-01-06	4	16
2020-01-07	5	25
2020-01-08	6	36
2020-01-09	7	49
2020-01-10	8	64
2020-01-13	9	81
2020-01-14	10	100

which is equivalent to `df.loc[:, ['x', 'x^2']]`. We can conclude here that there is equivalence between

`df[column_indexer]`

and

`df.loc[:, column_indexer]`

as both return all the rows and the selected columns. Good to know.

2.3.4 Selecting rows with rows and columns *positional indexes* : `.iloc[]`

In this section we describe how you can select specific rows and columns using their position in the table. You can use the `.iloc[ row positional indexer, column positional indexer ]` access operator using numeric positional indexes to identify rows and columns.

Access operator `.iloc[]` is somehow more limited than `.loc[]`. Still you have several options for both the `row positional indexer`, before the comma, and the `column positional indexer`, after the comma. We review them in the next sub-sections which parallel the different cases examined for `.loc[]`.

Of course you can combine different kind of indexers (single,List, slice, conditonal) for row and col indexers separately.

[146]:
```
df
```

[146]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024

```
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

Of course you can combine different kind of indexers (single,List, slice, conditonal) for row and col indexers separately.

Access operator `.iloc[]` is somehow more limited than `.loc[]`. Still you have several options for both the `row positional indexer`, before the comma, and the `column positional indexer`, after the comma.

2.3.4.1. Single positional index A single positional index label can be used to get one row and/or column only:

row indexer	col indexer	syntax/example	meaning
position	position `df.iloc[1, 3]`	`df.iloc[i,m]`	element in col position `m` at row position `i`

Let's select the single element in column 4th column and at 2nd row

[147]:
```
df.iloc[1,3]
```

[147]:
```
16
```

Accessing using an indexes out of bounds, raises a `IndexError`

[148]:
```
# raises IndexError as 100 is out of index positional bounds
#
# df.iloc[100,3]
```

[149]:
```
# raises IndexError as 10 is out of columns positional bounds
#
# df.iloc[1,10]
```

2.3.4.2. List of positional indexes a List of positional indexes can be used to get a List of rows and/or columnsç

row indexer	col indexer	syntax/example	meaning
…	positionsList	`df.iloc[..., [listOfColsPos]]`	elements in cols positions `[listOfColsPos]` at rows…
	`df.iloc[..., [1,2,4]]`		

row indexer	col indexer	syntax/example	meaning
positionsList	...	`df.iloc[[listOfRowsPos], ...]`	elements at rows positions `[listOfRowsPos]` in cols...
	`df.iloc[[0,3,5], ...]`		

SelectingLists of rows and/or cols according to their positions is straightforward

[150]:
```
df.iloc[[0,3,5], [1,2,4]]
```

[150]:
```
            x^2  x^3   x^5
2020-01-01    1    1     1
2020-01-06   16   64  1024
2020-01-08   36  216  7776
```

and works as expected. Other examples:

[151]:
```
df.iloc[0, [1,2,4]]
```

[151]:
```
x^2    1
x^3    1
x^5    1
Name: 2020-01-01 00:00:00, dtype: int32
```

[152]:
```
df.iloc[[0,3,5], 1]
```

[152]:
```
2020-01-01     1
2020-01-06    16
2020-01-08    36
Name: x^2, dtype: int32
```

2.3.4.3. Slice of positional indexes A slice of positional indexes can be used to get a slice of rows and/or columns:

row indexer	col indexer	syntax/example	meaning
slice	...	`df.iloc[i:j:k, ...]`	elements at rows as per slice `i:j:k` and in cols...
	`df.iloc[1:7:2, ...]`		
...	slice	`df.iloc[..., m:n:q]`	elements in cols as per slice `m:n:q` and at rows...
	`df.iloc[..., 1:3]`		

We can slice rows and/or columns. A few examples:

[153]:
```
df.iloc[1:7:2, 1:3]
```

[153]:
```
            x^2  x^3
2020-01-02    4    8
2020-01-06   16   64
2020-01-08   36  216
```

where the slice `1:7:2` selects all the rows from position 1 (included) to 7 (excluded), each 2 rows and the slice `1:3` selects each column from position 1 (included) to position 3 (excluded).

You can combine with other ways of positional indexing. Other examples:

[154]:
```
df.iloc[1, 1:3]
```

[154]:
```
x^2    4
x^3    8
Name: 2020-01-02 00:00:00, dtype: int32
```

[155]:
```
df.iloc[[2,3,4], 1:3]
```

[155]:
```
            x^2  x^3
2020-01-03    9   27
2020-01-06   16   64
2020-01-07   25  125
```

2.3.4.4. Conditional Expression to filter rows A logical conditional expression can be used to filter rows but it cannot be a NumPy array of booleans, not a Pandas Series:

row indexer	col indexer	syntax/example	meaning
logical condition (array)	…	`df.iloc[(logicalCondition).values, ...]` `df.iloc[df['x^2'].values > 5, ...]`	elements at rows filtered as per logical condition and in cols…

The use of `.iloc[]` with conditional expression is somehow nastier and I rarely use it. But still..

Let's go back to our $x^2 \leq x$ logical condition

[156]:
```
df_decimal
```

[156]:
```
                 x       x^2       x^3        x^4        x^5
0.000000  0.000000  0.000000  0.000000   0.000000   0.000000
0.222222  0.222222  0.049383  0.010974   0.002439   0.000542
0.444444  0.444444  0.197531  0.087791   0.039018   0.017342
```

```
0.666667  0.666667  0.444444  0.296296   0.197531   0.131687
0.888889  0.888889  0.790123  0.702332   0.624295   0.554929
1.111111  1.111111  1.234568  1.371742   1.524158   1.693509
1.333333  1.333333  1.777778  2.370370   3.160494   4.213992
1.555556  1.555556  2.419753  3.764060   5.855205   9.108097
1.777778  1.777778  3.160494  5.618656   9.988721  17.757727
2.000000  2.000000  4.000000  8.000000  16.000000  32.000000
```

[157]:
```python
df_decimal['x^2'] <= df_decimal['x']
```

[157]:
```
0.000000     True
0.222222     True
0.444444     True
0.666667     True
0.888889     True
1.111111    False
1.333333    False
1.555556    False
1.777778    False
2.000000    False
dtype: bool
```

You might be tempted to input that condition as a row indexer in the `.iloc[]` - as you do with the `.loc[]` access operator - and input a positional index as a column indexer. But this raises a `ValueError`

[158]:
```python
# raises ValueError
#
# df_decimal.iloc[df_decimal['x^2'] <= df_decimal['x'], [0,1]]
```

The fact is that the logical condition in the row indexer has to be an array of booleans... how can we transform it? `.values` on the rescue

[159]:
```python
(df_decimal['x^2'] <= df_decimal['x']).values
```

[159]:
```
array([ True,  True,  True,  True,  True, False, False, False, False,
       False])
```

[160]:
```python
df_decimal.iloc[(df_decimal['x^2'] <= df_decimal['x']).values, [0,1]]
```

[160]:

	x	x^2
0.000000	0.000000	0.000000
0.222222	0.222222	0.049383
0.444444	0.444444	0.197531
0.666667	0.666667	0.444444
0.888889	0.888889	0.790123

This is what we wanted! The `.values` attribute, returns the values of the boolean Series `df_decimal['x^2'] <= df_decimal['x']`, which is then a NumPy array of booleans. Which we can use a rows indexer in the `.iloc[]`.

En passant, notice that instead of `.values` the whole expression, you could have alternativelly use `.values` on each Pandas Series involved in the logical expression

[161]:
```
df_decimal.iloc[df_decimal['x^2'].values <= df_decimal['x'].values, [0,1]]
```

[161]:

	x	x^2
0.000000	0.000000	0.000000
0.222222	0.222222	0.049383
0.444444	0.444444	0.197531
0.666667	0.666667	0.444444
0.888889	0.888889	0.790123

Same same.

2.3.4.5. Colon : A colon : can be used to specify that you want to select all rows or columns:

row indexer	col indexer	syntax/example	meaning
:	:	`df.iloc[:, :]`	all the elements (equivalent to just `df`)
:	…	`df.iloc[:, ...]`	elements from all the rows and in cols…
…	:	`df.iloc[..., :]`	elements from all the cols and at rows…
…	omitted	`df.iloc[...]`	equivalent to `df.iloc[..., :]`

As you can notice this is exactly the same behavior that you have with `.loc[]`.

In particular, `df.iloc[:,:]` is equivalent to just typing `df`

[162]:
```
df.iloc[:,:]
```

[162]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[163]:
```
df
```

[163]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32

2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

If you want all rows but just some columns you can use the colon `:` as a row indexer

[164]:
```
df.iloc[:, [0,1]]
```

[164]:

	x	x^2
2020-01-01	1	1
2020-01-02	2	4
2020-01-03	3	9
2020-01-06	4	16
2020-01-07	5	25
2020-01-08	6	36
2020-01-09	7	49
2020-01-10	8	64
2020-01-13	9	81
2020-01-14	10	100

In the same way, if you want all the columns, but just a few rows, you can use the colon `:` as a cols indexer

[165]:
```
df.iloc[6:, :]
```

[165]:

	x	x^2	x^3	x^4	x^5
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Notice that if you want all the columns, you can completely omit the column indexer. Therefore, `df.iloc[6:, :]` is equivalent to `df.iloc[6:]`

[166]:
```
df.iloc[6:]
```

[166]:

	x	x^2	x^3	x^4	x^5
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

Finally, notice that - as already pointed out in the `.loc[]` case - if you omit the row indexer to ask for all the rows and few cols, you get the opposite

[167]:
```
df.iloc[[0,2]]
```

[167]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-03	3	9	27	81	243

that is, `[0,2]` gets interpreted as row positions. That's because if a single indexer is provided to `.iloc[]`, it gets interpreted as row indexer.

To get the `[0,2]` columns you have to use the colon `:` as row indexer

[168]:
```
df.iloc[:, [0,2]]
```

[168]:

	x	x^3
2020-01-01	1	1
2020-01-02	2	8
2020-01-03	3	27
2020-01-06	4	64
2020-01-07	5	125
2020-01-08	6	216
2020-01-09	7	343
2020-01-10	8	512
2020-01-13	9	729
2020-01-14	10	1000

Live with it.

2.4 Creating (and deleting) New Columns

Creating new columns is as simple as defining them

[169]:
```
df_refData
```

[169]:

	S&P Rating	Spread	Country
Firm_1	A	100	USA
Firm_2	BB	300	ITA
Firm_3	AA	70	UK
Firm_4	CCC	700	ITA

Suppose we want to add the share price (in USD) for each firm as an additional column `'Share Price'`

[170]:
```
df_refData['Share Price'] = [21.5, 15.0, 32.25, 2.5]
df_refData
```

[170]:

	S&P Rating	Spread	Country	Share Price
Firm_1	A	100	USA	21.50
Firm_2	BB	300	ITA	15.00
Firm_3	AA	70	UK	32.25

```
Firm_4         CCC     700     ITA          2.50
```

Easy, isn't it? New values, wrapped in theList, are assigned according to their input order to the corresponding index of `df_refData`.

Let's define the number of outstanding shares too, as new column `'Number of Shares'` (in billions)

[171]:
```python
df_refData['Number of Shares'] = [20, 3, 5, 2]
df_refData
```

[171]:

	S&P Rating	Spread	Country	Share Price	Number of Shares
Firm_1	A	100	USA	21.50	20
Firm_2	BB	300	ITA	15.00	3
Firm_3	AA	70	UK	32.25	5
Firm_4	CCC	700	ITA	2.50	2

We are now ready to compute the market capitalization `'Market Cap'` for each firm, as

`'Market Cap'` = `'Share Price'` × `'Number of Shares'`,

which serves here as an example of a column defined from two other existing columns

[172]:
```python
df_refData['Market Cap'] = df_refData['Share Price'] * df_refData['Number of␣
↪Shares']
df_refData
```

[172]:

	S&P Rating	Spread	Country	Share Price	Number of Shares	Market Cap
Firm_1	A	100	USA	21.50	20	430.00
Firm_2	BB	300	ITA	15.00	3	45.00
Firm_3	AA	70	UK	32.25	5	161.25
Firm_4	CCC	700	ITA	2.50	2	5.00

as you can see, each product of the share price and the number of shares is computed *elementwise*. This is a recurring feature, that we have already observed several times (e.g. NumPy arrays methods, Pandas Series methods, etc.).

What if we decide we want to keep only the `'Market Cap'`? You can delete columns as you would to delete `key: value` pairs from a Python Dict

[173]:
```python
del df_refData['Share Price']
df_refData
```

[173]:

	S&P Rating	Spread	Country	Number of Shares	Market Cap
Firm_1	A	100	USA	20	430.00
Firm_2	BB	300	ITA	3	45.00
Firm_3	AA	70	UK	5	161.25
Firm_4	CCC	700	ITA	2	5.00

[174]:
```python
del df_refData['Number of Shares']
df_refData
```

[174]:

	S&P Rating	Spread	Country	Market Cap
Firm_1	A	100	USA	430.00
Firm_2	BB	300	ITA	45.00
Firm_3	AA	70	UK	161.25
Firm_4	CCC	700	ITA	5.00

Notice that you can define new columns also according to logical conditions applied to other columns

[175]:
```
df_refData['isBlueChip'] = df_refData['Market Cap'] > 100.0
df_refData
```

[175]:

	S&P Rating	Spread	Country	Market Cap	isBlueChip
Firm_1	A	100	USA	430.00	True
Firm_2	BB	300	ITA	45.00	False
Firm_3	AA	70	UK	161.25	True
Firm_4	CCC	700	ITA	5.00	False

again, the logical condition that defines the new column `'isBlueChip'` works elementwise.

Finally, notice that the each column might have in principle, its own data-type

[176]:
```
df_refData.dtypes
```

[176]:
```
S&P Rating     object
Spread          int64
Country        object
Market Cap    float64
isBlueChip       bool
dtype: object
```

where `int64` and `float64` are for Integers and Floats, respectively, `object` is for Strings and `bool` is for boolean values.

2.5 Basic Analytics

As a direct generalization of Pandas Series, Pandas DataFrames too feature vectorized operations, a lot of built-in methods and can be safely passed to most of NumPy universal functions (that would expect NumPy arrays in input).

[177]:
```
df_decimal
```

[177]:

	x	x^2	x^3	x^4	x^5
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.222222	0.222222	0.049383	0.010974	0.002439	0.000542
0.444444	0.444444	0.197531	0.087791	0.039018	0.017342
0.666667	0.666667	0.444444	0.296296	0.197531	0.131687
0.888889	0.888889	0.790123	0.702332	0.624295	0.554929
1.111111	1.111111	1.234568	1.371742	1.524158	1.693509
1.333333	1.333333	1.777778	2.370370	3.160494	4.213992

```
1.555556  1.555556  2.419753  3.764060   5.855205   9.108097
1.777778  1.777778  3.160494  5.618656   9.988721  17.757727
2.000000  2.000000  4.000000  8.000000  16.000000  32.000000
```

2.5.1 *Vectorized* operations

DataFrame-Number and DataFrame-DataFrame operations are vectorized:

[178]:
```
df_decimal * 2
```

[178]:
```
                 x       x^2        x^3        x^4        x^5
0.000000  0.000000  0.000000   0.000000   0.000000   0.000000
0.222222  0.444444  0.098765   0.021948   0.004877   0.001084
0.444444  0.888889  0.395062   0.175583   0.078037   0.034683
0.666667  1.333333  0.888889   0.592593   0.395062   0.263374
0.888889  1.777778  1.580247   1.404664   1.248590   1.109858
1.111111  2.222222  2.469136   2.743484   3.048316   3.387018
1.333333  2.666667  3.555556   4.740741   6.320988   8.427984
1.555556  3.111111  4.839506   7.528121  11.710410  18.216193
1.777778  3.555556  6.320988  11.237311  19.977442  35.515453
2.000000  4.000000  8.000000  16.000000  32.000000  64.000000
```

[179]:
```
df_decimal + 10*df_decimal
```

[179]:
```
                  x        x^2        x^3         x^4         x^5
0.000000   0.000000   0.000000   0.000000    0.000000    0.000000
0.222222   2.444444   0.543210   0.120713    0.026825    0.005961
0.444444   4.888889   2.172840   0.965706    0.429203    0.190757
0.666667   7.333333   4.888889   3.259259    2.172840    1.448560
0.888889   9.777778   8.691358   7.725652    6.867246    6.104219
1.111111  12.222222  13.580247  15.089163   16.765737   18.628597
1.333333  14.666667  19.555556  26.074074   34.765432   46.353909
1.555556  17.111111  26.617284  41.404664   64.407255  100.189063
1.777778  19.555556  34.765432  61.805213  109.875934  195.334993
2.000000  22.000000  44.000000  88.000000  176.000000  352.000000
```

2.5.2 Built-in methods

There are tons of built-in methods. By default, built-in methods work column-wise:

[180]:
```
df_decimal.sum()
```

[180]:
```
x       10.000000
x^2     14.074074
x^3     22.222222
x^4     37.391861
x^5     65.477824
dtype: float64
```

which returns the Series of the sum of each column of `df_decimal`. This is equivalent to

[181]:
```
df_decimal.sum(axis=0)
```

[181]:
```
x        10.000000
x^2      14.074074
x^3      22.222222
x^4      37.391861
x^5      65.477824
dtype: float64
```

You can change the `axis` parameter to have the corresponding row-wise result

[182]:
```
df_decimal.sum(axis=1)
```

[182]:
```
0.000000      0.000000
0.222222      0.285559
0.444444      0.786127
0.666667      1.736626
0.888889      3.560568
1.111111      6.935088
1.333333     12.855967
1.555556     22.702671
1.777778     38.303375
2.000000     62.000000
dtype: float64
```

which returns a Series of the sums of each row of `df_decimal`.

2.5.3 Interoperability with NumPy's universal functions

Most of NumPy universal functions, which expect NumPy arrays in input, work with Pandas DataFrames in input as well

[183]:
```
np.exp(df_decimal)
```

[183]:

	x	x^2	x^3	x^4	x^5
0.000000	1.000000	1.000000	1.000000	1.000000e+00	1.000000e+00
0.222222	1.248849	1.050622	1.011034	1.002442e+00	1.000542e+00
0.444444	1.559623	1.218391	1.091760	1.039790e+00	1.017493e+00
0.666667	1.947734	1.559623	1.344869	1.218391e+00	1.140751e+00
0.888889	2.432425	2.203668	2.018454	1.866929e+00	1.741817e+00
1.111111	3.037732	3.436893	3.942212	4.591276e+00	5.438530e+00
1.333333	3.793668	5.916694	10.701355	2.358224e+01	6.762595e+01
1.555556	4.737718	11.243083	43.123166	3.490464e+02	9.028095e+03
1.777778	5.916694	23.582239	275.518752	2.177943e+04	5.153268e+07
2.000000	7.389056	54.598150	2980.957987	8.886111e+06	7.896296e+13

2.5.4 .groupby() category

Analytics can be performed on a per-group base, using the `.groupby()` method.

[184]:
```
df_decimal
```

[184]:

	x	x^2	x^3	x^4	x^5
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.222222	0.222222	0.049383	0.010974	0.002439	0.000542
0.444444	0.444444	0.197531	0.087791	0.039018	0.017342
0.666667	0.666667	0.444444	0.296296	0.197531	0.131687
0.888889	0.888889	0.790123	0.702332	0.624295	0.554929
1.111111	1.111111	1.234568	1.371742	1.524158	1.693509
1.333333	1.333333	1.777778	2.370370	3.160494	4.213992
1.555556	1.555556	2.419753	3.764060	5.855205	9.108097
1.777778	1.777778	3.160494	5.618656	9.988721	17.757727
2.000000	2.000000	4.000000	8.000000	16.000000	32.000000

Let's divide our DataFrame in two groups according to whether $x <= 1$ or $x > 1$. This can be easily achieved defining a column `'x range'` of Strings `'x > 1'` or `'x <= 1'` according to whether columns `'x'` have values greater or smaller-or-equal than 1.

[185]:
```
df_decimal['x range'] = ['x > 1' if x > 1 else 'x <= 1' for x in␣
↪df_decimal['x']]
df_decimal
```

[185]:

	x	x^2	x^3	x^4	x^5	x range
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	x <= 1
0.222222	0.222222	0.049383	0.010974	0.002439	0.000542	x <= 1
0.444444	0.444444	0.197531	0.087791	0.039018	0.017342	x <= 1
0.666667	0.666667	0.444444	0.296296	0.197531	0.131687	x <= 1
0.888889	0.888889	0.790123	0.702332	0.624295	0.554929	x <= 1
1.111111	1.111111	1.234568	1.371742	1.524158	1.693509	x > 1
1.333333	1.333333	1.777778	2.370370	3.160494	4.213992	x > 1
1.555556	1.555556	2.419753	3.764060	5.855205	9.108097	x > 1
1.777778	1.777778	3.160494	5.618656	9.988721	17.757727	x > 1
2.000000	2.000000	4.000000	8.000000	16.000000	32.000000	x > 1

Before going forward, just notice how we usedList comprehension to define the new `'x range'` column of Strings. What does

```
['x > 1' if x > 1 else 'x <= 1' for x in df_decimal['x']]
```

do? (try to answer yourself before reading below)

Easy: it is a List, as is surrounded by square brackets `[]`. Ok, a List of what? Well, the expression

```
for x in df_decimal['x']
```

defines a loop over the values in column `'x'` of `df_decimal`, which at each loop iteration gives the dummy name `x` to value considered at that iteration. Ok, so? Well, each value `x` is then checked whether it is `x > 1` or not. Good… so how is theList filled then? Well, theList is filled with String `'x > 1'` at iterations in which the dummy variable satisfies condition `x > 1` and with String `'x <= 1'` otherwise. Bravo! :)

Now that we have this grouping column `'x range'`, we can compute `.sum()`, `.mean()`, `.std()` and whatever you want on the two seperate groups just prepending the grouping condition `.groupby('x range')` to the method that you want to use

[186]:
```python
df_decimal.groupby("x range").sum()
```

[186]:
```
                x         x^2        x^3        x^4        x^5
x range
x <= 1   2.222222    1.481481   1.097394   0.863283   0.704500
x > 1    7.777778   12.592593  21.124829  36.528578  64.773324
```

and this returns the `.sum()` over columns, separately for each group of rows, according to the grouping defined by `'x range'`.

Say we were interested in the group `.mean()` of `'x^5'` column only. Well easy, just use the `[]` access operator after the grouping statement

[187]:
```python
df_decimal.groupby('x range')['x^5'].sum()
```

[187]:
```
x range
x <= 1     0.704500
x > 1     64.773324
Name: x^5, dtype: float64
```

You can count the number of rows in each group: `.size()`

[188]:
```python
df_decimal.groupby('x range').size()
```

[188]:
```
x range
x <= 1    5
x > 1     5
dtype: int64
```

You can group by more than group simultaneously… Let's define another group, defining a new column `'x over x^2'`, which marks the rows according to whether $x > x^2$, $x = x^2$ or $x < x^2$

[189]:
```python
df_decimal['x over x^2'] = ['x > x^2' if x > x2 else 'x = x^2' if x == x2 else␣
 ↪'x < x^2'
                            for (x,x2) in zip(df_decimal['x'],␣
 ↪df_decimal['x^2'])]
df_decimal
```

[189]:
```
                 x       x^2       x^3        x^4        x^5 x range  \
0.000000  0.000000  0.000000  0.000000   0.000000   0.000000  x <= 1
0.222222  0.222222  0.049383  0.010974   0.002439   0.000542  x <= 1
```

```
0.444444  0.444444  0.197531  0.087791   0.039018   0.017342  x <= 1
0.666667  0.666667  0.444444  0.296296   0.197531   0.131687  x <= 1
0.888889  0.888889  0.790123  0.702332   0.624295   0.554929  x <= 1
1.111111  1.111111  1.234568  1.371742   1.524158   1.693509   x > 1
1.333333  1.333333  1.777778  2.370370   3.160494   4.213992   x > 1
1.555556  1.555556  2.419753  3.764060   5.855205   9.108097   x > 1
1.777778  1.777778  3.160494  5.618656   9.988721  17.757727   x > 1
2.000000  2.000000  4.000000  8.000000  16.000000  32.000000   x > 1

         x over x^2
0.000000    x = x^2
0.222222    x > x^2
0.444444    x > x^2
0.666667    x > x^2
0.888889    x > x^2
1.111111    x < x^2
1.333333    x < x^2
1.555556    x < x^2
1.777778    x < x^2
2.000000    x < x^2
```

How the hell did you define the `'x over x^2'` column? Well, it's a bit harder, but not that much…

The first thing you need to understand is a the built-in function `zip()`.

It works as follows: you have to use `zip()` in a loop when you want to iterate over elements of several sequences at the same time. An example: say we want to prints the pairs of elements coming from twoLists of the same length

[190]:
```
a = [1,2,3,4,5]

for ai in a:
    print(ai)
```

```
1
2
3
4
5
```

[191]:
```
a = [1,2,3,4,5]
b = [6,7,8,9,10]

for (ai, bi) in zip(a,b):
    print(ai, bi)
```

```
1 6
2 7
3 8
```

```
4 9
5 10
```

So, as you can see, `zip()` in a loop iterate over the pairs of elements (a Tuple, whose elements we called `(ai, bi)`), one from eachList. It's not limited to just twoLists of course

[192]:
```
a = [1,2,3,4,5]
b = [6,7,8,9,10]
c = [11, 12, 13, 14, 15]

for (ai, bi, ci) in zip(a,b,c):
    print(ai, bi, ci)
```

```
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15
```

So now what

```
for (x,x2) in zip(df_decimal['x'], df_decimal['x^2'])
```

is, it's a bit more clear: it defines a loop over the values in the pair of columns `'x'` and `'x^2'` a of `df_decimal` and, at each loop iteration, it gives the dummy names `x` and `x2` to the elements of the Tuple returned by `zip()`.

Now, the rest of the expression

```
'x > x^2' if x > x2 else 'x = x^2' if x == x2 else 'x < x^2'
```

is simply the specification of the values that we want to have in column `'x over x^2'`: depending on the values of the dummy variables `x` and `x2`, the column is filled with `'x > x^2'`, `'x = x^2'` or `'x < x^2'`. A pseudo-code for this is:

```
if x > x2:
    # fill with value: 'x > x^2'
    else:
        if x == x2:
            # fill with value 'x = x^2'
        else:
            # fill with value 'x < x^2'
```

Ok, eonough.

So we now have two columns, `'x range'` and `'x over x^2'`, that represent possible groupings for our DataFrame

[193]:
```
df_decimal
```

[193]:

```
                     x       x^2       x^3        x^4        x^5 x range  \
0.000000  0.000000  0.000000  0.000000   0.000000   0.000000   x <= 1
0.222222  0.222222  0.049383  0.010974   0.002439   0.000542   x <= 1
```

```
0.444444  0.444444  0.197531  0.087791   0.039018   0.017342  x <= 1
0.666667  0.666667  0.444444  0.296296   0.197531   0.131687  x <= 1
0.888889  0.888889  0.790123  0.702332   0.624295   0.554929  x <= 1
1.111111  1.111111  1.234568  1.371742   1.524158   1.693509   x > 1
1.333333  1.333333  1.777778  2.370370   3.160494   4.213992   x > 1
1.555556  1.555556  2.419753  3.764060   5.855205   9.108097   x > 1
1.777778  1.777778  3.160494  5.618656   9.988721  17.757727   x > 1
2.000000  2.000000  4.000000  8.000000  16.000000  32.000000   x > 1

         x over x^2
0.000000    x = x^2
0.222222    x > x^2
0.444444    x > x^2
0.666667    x > x^2
0.888889    x > x^2
1.111111    x < x^2
1.333333    x < x^2
1.555556    x < x^2
1.777778    x < x^2
2.000000    x < x^2
```

an analytics, like `.sum()` over compound groups can be done in the same way as before, just wrap the grouping columns as a List in the `.groupby()` method

[194]:
```python
df_decimal.groupby(['x range', 'x over x^2']).sum()
```

[194]:
```
                           x        x^2        x^3        x^4        x^5
x range x over x^2
x <= 1  x = x^2     0.000000   0.000000   0.000000   0.000000   0.000000
        x > x^2     2.222222   1.481481   1.097394   0.863283   0.704500
x > 1   x < x^2     7.777778  12.592593  21.124829  36.528578  64.773324
```

that shows column-wise sums divided by the double criterion $x > 1/x \leq 1$ and $x > x^2$, $x = x^2$ or $x < x^2$.

And, as before you can count the rows in each group

[195]:
```python
df_decimal.groupby(['x range', 'x over x^2']).size()
```

[195]:
```
x range  x over x^2
x <= 1   x = x^2       1
         x > x^2       4
x > 1    x < x^2       5
dtype: int64
```

and can filter just few columns if you want, as before

[196]:
```python
df_decimal.groupby(['x range', 'x over x^2'])['x^5'].sum()
```

```
[196]: x range   x over x^2
       x <= 1    x = x^2         0.000000
                 x > x^2         0.704500
       x > 1     x < x^2        64.773324
       Name: x^5, dtype: float64
```

2.6 Data Alignment

Talking about Pandas Series, we have observed that two different Series, when combined, gets aligned according to their indexes. As a direct extension, Pandas DataFrames consider not only rows but also columns to align data when combining to DataFrames.

```
[197]: df
```

```
[197]:
             x  x^2   x^3    x^4     x^5
2020-01-01   1    1     1      1       1
2020-01-02   2    4     8     16      32
2020-01-03   3    9    27     81     243
2020-01-06   4   16    64    256    1024
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

```
[198]: df1 = df.loc[:'2020-01-08', ['x', 'x^2', 'x^3']]
       df1
```

```
[198]:
            x  x^2  x^3
2020-01-01  1    1    1
2020-01-02  2    4    8
2020-01-03  3    9   27
2020-01-06  4   16   64
2020-01-07  5   25  125
2020-01-08  6   36  216
```

```
[199]: df2 = df.loc['2020-01-04':, ['x^3', 'x^4', 'x^5']]
       df2
```

```
[199]:
            x^3   x^4    x^5
2020-01-06   64   256   1024
2020-01-07  125   625   3125
2020-01-08  216  1296   7776
2020-01-09  343  2401  16807
2020-01-10  512  4096  32768
2020-01-13  729  6561  59049
```

```
2020-01-14   1000   10000   100000
```

As we see, `df1` and `df2` share: - indexes from `'2020-01-06'` to `'2020-01-08'` - column `'x^3'`

how, then, an operation as simple as `df1 + df2` is defined?

[200]:
```
df_1plus2 = df1 + df2
df_1plus2
```

[200]:
```
              x  x^2    x^3  x^4  x^5
2020-01-01  NaN  NaN    NaN  NaN  NaN
2020-01-02  NaN  NaN    NaN  NaN  NaN
2020-01-03  NaN  NaN    NaN  NaN  NaN
2020-01-06  NaN  NaN  128.0  NaN  NaN
2020-01-07  NaN  NaN  250.0  NaN  NaN
2020-01-08  NaN  NaN  432.0  NaN  NaN
2020-01-09  NaN  NaN    NaN  NaN  NaN
2020-01-10  NaN  NaN    NaN  NaN  NaN
2020-01-13  NaN  NaN    NaN  NaN  NaN
2020-01-14  NaN  NaN    NaN  NaN  NaN
```

As noticed for indexes only in Pandas Series, combined DataFrame will have the *union* of both indexes and columns, with operations performed elementwise and only where the `[row, column]` is shared by both DataFrames (here `['2020-01-06', 'x^3']`, `['2020-01-07', 'x^3']` and `['2020-01-08', 'x^3']`) and putting a `NaN` elsewhere.

Of course this generalizes to any other operation:

[201]:
```
df_1times2 = df1 * df2
df_1times2
```

[201]:
```
              x  x^2      x^3  x^4  x^5
2020-01-01  NaN  NaN      NaN  NaN  NaN
2020-01-02  NaN  NaN      NaN  NaN  NaN
2020-01-03  NaN  NaN      NaN  NaN  NaN
2020-01-06  NaN  NaN   4096.0  NaN  NaN
2020-01-07  NaN  NaN  15625.0  NaN  NaN
2020-01-08  NaN  NaN  46656.0  NaN  NaN
2020-01-09  NaN  NaN      NaN  NaN  NaN
2020-01-10  NaN  NaN      NaN  NaN  NaN
2020-01-13  NaN  NaN      NaN  NaN  NaN
2020-01-14  NaN  NaN      NaN  NaN  NaN
```

As is the case for Pandas Series, most basic analytics still works disregarding NaNs. That is, `NaN` are not counted.

[202]:
```
df_1plus2 ** 2
```

```
[202]:               x  x^2        x^3  x^4  x^5
       2020-01-01 NaN  NaN        NaN  NaN  NaN
       2020-01-02 NaN  NaN        NaN  NaN  NaN
       2020-01-03 NaN  NaN        NaN  NaN  NaN
       2020-01-06 NaN  NaN    16384.0  NaN  NaN
       2020-01-07 NaN  NaN    62500.0  NaN  NaN
       2020-01-08 NaN  NaN   186624.0  NaN  NaN
       2020-01-09 NaN  NaN        NaN  NaN  NaN
       2020-01-10 NaN  NaN        NaN  NaN  NaN
       2020-01-13 NaN  NaN        NaN  NaN  NaN
       2020-01-14 NaN  NaN        NaN  NaN  NaN
```

```
[203]: df_1plus2.sum(axis=0) #columns-wise sum
```

```
[203]: x        0.0
       x^2      0.0
       x^3    810.0
       x^4      0.0
       x^5      0.0
       dtype: float64
```

```
[204]: df_1plus2.sum(axis=1) #row-wise sum
```

```
[204]: 2020-01-01      0.0
       2020-01-02      0.0
       2020-01-03      0.0
       2020-01-06    128.0
       2020-01-07    250.0
       2020-01-08    432.0
       2020-01-09      0.0
       2020-01-10      0.0
       2020-01-13      0.0
       2020-01-14      0.0
       Freq: B, dtype: float64
```

```
[205]: df_1plus2.mean(axis=0) # column-wise mean
```

```
[205]: x        NaN
       x^2      NaN
       x^3    270.0
       x^4      NaN
       x^5      NaN
       dtype: float64
```

```
[206]: df_1plus2.mean(axis=1) # row-wise mean
```

[206]:
```
2020-01-01      NaN
2020-01-02      NaN
2020-01-03      NaN
2020-01-06    128.0
2020-01-07    250.0
2020-01-08    432.0
2020-01-09      NaN
2020-01-10      NaN
2020-01-13      NaN
2020-01-14      NaN
Freq: B, dtype: float64
```

[207]:
```
df_1plus2.std(axis=0) # column-wise standard deviation
```

[207]:
```
x             NaN
x^2           NaN
x^3    152.983659
x^4           NaN
x^5           NaN
dtype: float64
```

[208]:
```
df_1plus2.std(axis=1) # row-wise standard deviation
```

[208]:
```
2020-01-01   NaN
2020-01-02   NaN
2020-01-03   NaN
2020-01-06   NaN
2020-01-07   NaN
2020-01-08   NaN
2020-01-09   NaN
2020-01-10   NaN
2020-01-13   NaN
2020-01-14   NaN
Freq: B, dtype: float64
```

2.7 Combine data from multiple DataFrames

In this section we explore two ways to combine two DataFrames: concatenating or joining them. Let's work with our `df` DataFrame

[209]:
```
df
```

[209]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024

```
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

which features 10 rows and 5 columns

[210]:
```python
df.shape
```

[210]:
```
(10, 5)
```

2.7.1 Concatenating DataFrames: `pd.concat()`

Function

`pd.concat([df1, df2,...], axis=0, sort=True)`

concatenates theList of (two or more) DataFrames according to the `axis`:

- 0, default, for rows-wise concatenation (that is, vertically), or
- 1, for column-wise concatenation (that is, horizontally)

and sorts (if `sort=True`, default) or leaves untouched (if `sort=False`) the non-concatenating axis, that is

- columns in case of vertical concatenation, or
- rows, in case of horizontal concatenation.

2.7.Vertical concatenation: `pd.concat(...[, axis=0])` We start considering *vertical* concatenation. We define two DataFrames, considering two - non overlapping - slices of rows of the `df` DataFrame

[211]:
```python
df_up = df.loc[:'2020-01-08']
df_up
```

[211]:
```
            x  x^2  x^3   x^4   x^5
2020-01-01  1    1    1     1     1
2020-01-02  2    4    8    16    32
2020-01-03  3    9   27    81   243
2020-01-06  4   16   64   256  1024
2020-01-07  5   25  125   625  3125
2020-01-08  6   36  216  1296  7776
```

[212]:
```python
df_up.shape
```

[212]:
```
(6, 5)
```

[213]:
```python
df_down = df.loc['2020-01-09':]
df_down
```

[213]:

	x	x^2	x^3	x^4	x^5
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[214]:
```python
df_down.shape
```

[214]: (4, 5)

Notice that `df_up` and `df_down` share the same columns and do not have overlapping indexes. Let's concatenate them one over the other

[215]:
```python
df_up_down = pd.concat([df_up, df_down])
df_up_down
```

[215]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[216]:
```python
df_up_down.shape
```

[216]: (10, 5)

As expected, `df_up_down` is simply given by the vertical stacking of `df_up` and `df_down`, with columns of the same name matched.

Let's now keep the same columns, but introducing an overlap in the rows

[217]:
```python
df_up_overlap = df.loc[:'2020-01-08']
df_up_overlap
```

[217]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024

```
2020-01-07  5   25  125   625  3125
2020-01-08  6   36  216  1296  7776
```

[218]:
```python
df_up_overlap.shape
```

[218]:
```
(6, 5)
```

[219]:
```python
df_down_overlap = df.loc['2020-01-04':]
df_down_overlap
```

[219]:

	x	x^2	x^3	x^4	x^5
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[220]:
```python
df_down_overlap.shape
```

[220]:
```
(7, 5)
```

[221]:
```python
df_up_down_overlap = pd.concat([df_up_overlap, df_down_overlap])
df_up_down_overlap
```

[221]:

	x	x^2	x^3	x^4	x^5
2020-01-01	1	1	1	1	1
2020-01-02	2	4	8	16	32
2020-01-03	3	9	27	81	243
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-06	4	16	64	256	1024
2020-01-07	5	25	125	625	3125
2020-01-08	6	36	216	1296	7776
2020-01-09	7	49	343	2401	16807
2020-01-10	8	64	512	4096	32768
2020-01-13	9	81	729	6561	59049
2020-01-14	10	100	1000	10000	100000

[222]:
```python
df_up_down_overlap.shape
```

[222]:
```
(13, 5)
```

Notice that indexes are simply stacked one over the other, such that the resulting `df_up_down_overlap` DataFrame has repeated indexes: '2020-01-06', '2020-01-07' and '2020-01-08'.

2.7.1.2. Horizontal concatenation: `pd.concat(..., axis=1)`

We now consider *horizontal* concatenation. We consider `df_refData` DataFrame and another DataFrame `df_otherRefData` with additional reference data for some firms

[223]:
```
df_refData
```

[223]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip
Firm_1          A     100     USA      430.00        True
Firm_2         BB     300     ITA       45.00       False
Firm_3         AA      70      UK      161.25        True
Firm_4        CCC     700     ITA        5.00       False
```

[224]:
```
df_otherRefData = pd.DataFrame(
                    data={
                        "Moody's Rating": ['A2', 'Ba2', 'Aa2', 'Caa2'], #␣
 ↪notice use of "" to allow apostrophe ' in String
                        'Fitch Rating': ['A', 'BB', 'AA', 'CCC']
                    },
                    index=['Firm_1', 'Firm_2', 'Firm_3', 'Firm_4'])

df_otherRefData
```

[224]:
```
       Moody's Rating Fitch Rating
Firm_1             A2            A
Firm_2            Ba2           BB
Firm_3            Aa2           AA
Firm_4           Caa2          CCC
```

[225]:
```
df_otherRefData.shape
```

[225]: (4, 2)

Notice that `df_otherRefData` and `df_refData` share the same rows and do not have overlapping columns. Let's concatenate them one next to the other

[226]:
```
df_completeRefData = pd.concat([df_refData, df_otherRefData], axis=1)
df_completeRefData
```

[226]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip Moody's Rating  \
Firm_1          A     100     USA      430.00        True             A2
Firm_2         BB     300     ITA       45.00       False            Ba2
Firm_3         AA      70      UK      161.25        True            Aa2
Firm_4        CCC     700     ITA        5.00       False           Caa2

       Fitch Rating
Firm_1            A
Firm_2           BB
Firm_3           AA
```

```
Firm_4          CCC
```

[227]:
```
df_completeRefData.shape
```

[227]:
```
(4, 7)
```

As expected, `df_completeRefData` is simply given by the horizontal stacking of `df_refData` and `df_otherRefData`, with common indexes matched.

Let's now keep the same indexes, but introducing an overlap in the columns, such that `'Spread'` column is shared by both `df_refData` and `df_otherRefData_overlap`

[228]:
```
df_otherRefData_overlap = pd.DataFrame(
                             data={
                                   "Moody's Rating": ['A2', 'Ba2', 'Aa2',␣
 ↪'Caa2'],
                                   'Fitch Rating': ['A', 'BB', 'AA', 'CCC'],
                                   'Spread': [100, 300, 70, 700]
                                  },
                             index=['Firm_1', 'Firm_2', 'Firm_3', 'Firm_4'])

df_otherRefData_overlap
```

[228]:
```
       Moody's Rating Fitch Rating  Spread
Firm_1             A2            A     100
Firm_2            Ba2           BB     300
Firm_3            Aa2           AA      70
Firm_4           Caa2          CCC     700
```

[229]:
```
df_otherRefData_overlap.shape
```

[229]:
```
(4, 3)
```

[230]:
```
df_completeRefData_overlap = pd.concat([df_refData, df_otherRefData_overlap],␣
 ↪axis=1)
df_completeRefData_overlap
```

[230]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip Moody's Rating  \
Firm_1          A     100     USA      430.00        True             A2
Firm_2         BB     300     ITA       45.00       False            Ba2
Firm_3         AA      70      UK      161.25        True            Aa2
Firm_4        CCC     700     ITA        5.00       False           Caa2

       Fitch Rating  Spread
Firm_1            A     100
Firm_2           BB     300
Firm_3           AA      70
Firm_4          CCC     700
```

[231]:
```
df_otherRefData_overlap.shape
```

[231]: (4, 3)

Notice that columns are simply stacked one next to the other, such that the resulting `df_otherRefData_overlap` DataFrame has repeated column: `'Spread'`.

We have only scratched the surface of the concatenation possibilities offered by Pandas, a good further reading can be found in the user guide: Concatenating objects.

Take-home message: we typically use concatenation when we have two DataFrames that are clearly complementary. For example,

- you might need to *vertically* concatenate datasets of the same kind of data that you have downloaded sequentially and that thus have consecutive and non-overallaping Dates indexes, like `df_up` and `df_down` in our example. In this case it may be useful to vertically stack the two, to have a longer historical timeSeries of data.
- you might need to *horizontally* concatenate two reference data datasets, both containing reference data for the same set of entities, but one containing some reference data, like `'S&P Rating'`, and the other one, complementary reference data, like `"Moody's Rating"` and `'Fitch Rating'`. And thus the two sets of columns are non-overlapping. In this case it may be useful to horizontally stack the two, to have a more complete dataset for each instrument.

If you have overlaps in indexes and/or columns, you probably should be looking for a `.join()` of the two DataFrames…

2.7.2 Joining DataFrames: `.join()` and `pd.merge()`

Pandas has a lot of built-in functionalities to do in-memory join operations between datasets, in most aspects similiar to relational databases like SQL. Function `pd.merge()` is the the most general entry point for all standard database join operations between DataFrames (or Series). The related `.join()` method of a DataFrame object (which uses `pd.merge()` internally) is, instead, more specifically oriented toward index-on-index and column(s)-on-index join.

In this section we analyze a couple of common join situations you may encounter in your daily activities and how to tackle them.

2.7.2.1. Index-on-index join: `df1.join(df2[,how])` `.join()` is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. A good further reading can be found in the the user guide: Joining on index.

If you have two DataFrames, say `df1` and `df2`, you can combine them into a single DataFrame using method `join()` applied to `df1` and providing `df2` in input. Basic syntax is

```
df1.join(df2, how='left')
```

which joins `df1` with `df2` on the base of their indexes. Parameter `how` can take the following values:

- `'left'`, default, which uses `df1` indexes to do a *left-join* on `df2` indexes, or
- `'right'`, which uses `df2` indexes to do a left-join on `df1` indexes (equivalent to `df2.join(df1, how='left')`), or

- 'inner', which uses indexes in common (the intersection of) between df1 and df2 to do an *inner-join* of df1 and df2 indexes, or
- 'outer', which uses both indexes of (the union of) df1 and df2 to do an *outer-join* of df1 and df2 indexes.

there is another relevant parameter, on, which can be omitted in an index-on-index situation and that we will introduce in the next section when talking about index-on-column join.

We consider our DataFrame df_refData which stores some reference data information for Firm_1, Firm_2, Firm_3 and Firm_4, which are used as indexes.

[232]:
```
df_refData
```

[232]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip
Firm_1          A     100     USA      430.00        True
Firm_2         BB     300     ITA       45.00       False
Firm_3         AA      70      UK      161.25        True
Firm_4        CCC     700     ITA        5.00       False
```

We then define another DataFrame, df_otherRefData, can be used to complement df_refData informations on Moody's and Fitch ratings.

[233]:
```
df_otherRefData = pd.DataFrame(
                      data={
                          "Moody's Rating": ['Aa2', 'Caa2', 'A2', 'Aaa', 'Aa3'],␣
↪# "" allows use of apostrophe ' in String
                          'Fitch Rating': ['AA', 'CCC', 'A', 'AAA', 'AA-']
                      },
                      index=['Firm_3', 'Firm_4', 'Firm_1', 'Firm_5', 'Firm_6'])

df_otherRefData
```

[233]:
```
       Moody's Rating Fitch Rating
Firm_3            Aa2           AA
Firm_4           Caa2          CCC
Firm_1             A2            A
Firm_5            Aaa          AAA
Firm_6            Aa3          AA-
```

Notice that df_otherRefData has informations on: - Firm_1, Firm_3, Firm_4, indexes shared with df_refData; - new firms Firm_5 and Firm_6; - but doesn't have data for Firm_2.

We now consider a left-join of df_refData indexes on df_otherRefData. In other words, we want to keep all the rows of df_refData and complete the columns them with informations included in df_otherRefData, when available, on the bases of their shared indexes.

[234]:
```
df_refData.join(df_otherRefData, how='left')
```

[234]:
```
        S&P Rating  Spread Country  Market Cap  isBlueChip Moody's Rating  \
Firm_1           A     100     USA      430.00        True             A2
Firm_2          BB     300     ITA       45.00       False            NaN
Firm_3          AA      70      UK      161.25        True            Aa2
Firm_4         CCC     700     ITA        5.00       False           Caa2

       Fitch Rating
Firm_1            A
Firm_2          NaN
Firm_3           AA
Firm_4          CCC
```

Notice that the DataFrame returned has all - and only - the indexes (and rows) of `df_refData`, with data completed with `'Moody's' Rating` and `'Fitch Rating'` additional columns, coming from `df_otherRefData`. Data for `Firm_2`, which are missing in `df_otherRefData`, are marked as `NaN` in the output DataFrame.

Notice also that `how` parameter is `how='left'` by default and then can be omitted in the case of a left-join operation as this one

[235]:
```python
df_refData.join(df_otherRefData)
```

[235]:
```
        S&P Rating  Spread Country  Market Cap  isBlueChip Moody's Rating  \
Firm_1           A     100     USA      430.00        True             A2
Firm_2          BB     300     ITA       45.00       False            NaN
Firm_3          AA      70      UK      161.25        True            Aa2
Firm_4         CCC     700     ITA        5.00       False           Caa2

       Fitch Rating
Firm_1            A
Firm_2          NaN
Firm_3           AA
Firm_4          CCC
```

A right-join is the symmetric case of the left-join, where rows from `df_otherRefData` are kept, with data completed with those from `df_refData`. This can be achieved either using `how='right'`

[236]:
```python
df_refData.join(df_otherRefData, how='right')
```

[236]:
```
        S&P Rating  Spread Country  Market Cap isBlueChip Moody's Rating  \
Firm_3          AA    70.0      UK      161.25       True            Aa2
Firm_4         CCC   700.0     ITA        5.00      False           Caa2
Firm_1           A   100.0     USA      430.00       True             A2
Firm_5         NaN     NaN     NaN         NaN        NaN            Aaa
Firm_6         NaN     NaN     NaN         NaN        NaN            Aa3

       Fitch Rating
Firm_3           AA
```

```
Firm_4          CCC
Firm_1            A
Firm_5          AAA
Firm_6          AA-
```

or, as expected, inverting the role of the two DataFrames and doing a left-join

[237]:
```
df_otherRefData.join(df_refData, how='left')
```

[237]:
```
       Moody's Rating Fitch Rating S&P Rating  Spread Country  Market Cap  \
Firm_3            Aa2           AA         AA    70.0      UK      161.25
Firm_4           Caa2          CCC        CCC   700.0     ITA        5.00
Firm_1             A2            A          A   100.0     USA      430.00
Firm_5            Aaa          AAA        NaN     NaN     NaN         NaN
Firm_6            Aa3          AA-        NaN     NaN     NaN         NaN

       isBlueChip
Firm_3       True
Firm_4      False
Firm_1       True
Firm_5        NaN
Firm_6        NaN
```

As you can see, the output includes all the rows from `df_otherRefData`, with data completed with those from `df_refData`. Moreover, data for `Firm_5` and `Firm_6`, which are missing in `df_refData`, are marked as `NaN` in the output DataFrame.

We now consider an inner-join between `df_refData` and `df_otherRefData` indexes. In other words, we want to keep only the rows in common (on the bases of their shared indexes) between `df_refData` and `df_otherRefData`, completing columns from `df_refData` with those from `df_otherRefData`.

[238]:
```
df_refData.join(df_otherRefData, how='inner')
```

[238]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip Moody's Rating  \
Firm_1          A     100     USA      430.00        True             A2
Firm_3         AA      70      UK      161.25        True            Aa2
Firm_4        CCC     700     ITA        5.00       False           Caa2

       Fitch Rating
Firm_1            A
Firm_3           AA
Firm_4          CCC
```

Notice that the DataFrame returned has only the indexes which `df_refData` and `df_otherRefData` have in common. Data from the two DataFrames are integrated adding `'Moody's' Rating` and `'Fitch Rating'` columns, coming from `df_otherRefData`, to those from `df_refData`. Row for `Firm_2`, which is part of `df_refData` but is missing in `df_otherRefData`, is excluded from the output. In the same way, data for `Firm_5` and `Firm_6`, which are part of `df_refData` but missing in `df_otherRefData`, are excluded from the output.

We now consider a outer-join of `df_refData` indexes on `df_otherRefData`. In other words, we want to keep all the rows of `df_refData` and complete data with those included in `df_otherRefData`, when available, on the base of their shared indexes.

[239]:
```
df_refData.join(df_otherRefData, how='outer')
```

[239]:
```
       S&P Rating  Spread Country  Market Cap isBlueChip Moody's Rating  \
Firm_1          A   100.0     USA      430.00       True             A2
Firm_2         BB   300.0     ITA       45.00      False            NaN
Firm_3         AA    70.0      UK      161.25       True            Aa2
Firm_4        CCC   700.0     ITA        5.00      False           Caa2
Firm_5        NaN     NaN     NaN         NaN        NaN            Aaa
Firm_6        NaN     NaN     NaN         NaN        NaN            Aa3

       Fitch Rating
Firm_1            A
Firm_2          NaN
Firm_3           AA
Firm_4          CCC
Firm_5          AAA
Firm_6          AA-
```

The output DataFrames has all the rows from the two DataFrames (you see both `Firm_2` and `Firm_5` and `Firm_6`) and data, missing in one of the other original DataFrame, are marked with `NaN`.

2.7.2.2. Join on Column(s): `df1.join(df2,on)` and `pd.merge(df1, df2, on)` There are situations in which you want to join two DataFrames on the base of a key-column of one DataFrame and the index or another key-column of the other DataFrame. A good further reading can be found in the the user guide: Joining on index.

In this case you have to use the `on` parameter of `.join()` method

```
df1.join(df2, how='left', on=None)
```

You have to give to parameter `on`, which is left unspecified by default (`None` is the Python key-word to say that something is un-specified), the name of a column which is shared by `df1` and `df2` and on the base of which you want to perform the join. The `how` parameter keeps the same meaning and the idea is that you can do a left/righ/inner/outer-join on the base of the column shared and specified by `on`, instead of the indexes.

Let's complete `df_refData` with a `'Ticker'` column, which might represent the Bloomberg (or Reuters, or other data source) identifier for each firm.

[240]:
```
df_refData
```

[240]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip
Firm_1          A     100     USA      430.00        True
Firm_2         BB     300     ITA       45.00       False
Firm_3         AA      70      UK      161.25        True
```

```
Firm_4         CCC      700      ITA         5.00        False
```

[241]:
```
df_refData['Ticker'] = ['CDE', 'BCD', 'DEF', 'ABC']
df_refData
```

[241]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip Ticker
Firm_1          A     100     USA      430.00        True    CDE
Firm_2         BB     300     ITA       45.00       False    BCD
Firm_3         AA      70      UK      161.25        True    DEF
Firm_4        CCC     700     ITA        5.00       False    ABC
```

We now have another table `df_dividendInfo` which store some dividend informations. These informations are organized according to a List of ticker. Think of it as a big database where dividend informations for all the companies (not only those in your portfolio `df_refData`) are stored.

[242]:
```
df_dividendInfo = pd.DataFrame(
                    data={
                        'Dividend Yield (%)': [0.72, 0.45, 1.15, 0.96, 2.01, 1.
↪5, 0.3, 0.89],
                        'Dividend Frequency': ['Quarterly',
                                               'Monthly',
                                               'Biannually',
                                               'Quarterly',
                                               'Biannually',
                                               'Annually',
                                               'Monthly',
                                               'Quarterly']
                    },
                    index=['ABC', 'BCD', 'CDE', 'DEF', 'EFG', 'FGH', 'GHI',␣
↪'HIJ'])

df_dividendInfo
```

[242]:
```
     Dividend Yield (%) Dividend Frequency
ABC                0.72          Quarterly
BCD                0.45            Monthly
CDE                1.15         Biannually
DEF                0.96          Quarterly
EFG                2.01         Biannually
FGH                1.50           Annually
GHI                0.30            Monthly
HIJ                0.89          Quarterly
```

We want to complete our `df_refData` dataset with dividend informations. To do this, we can left-join `df_refData on` the `'Ticker'` key column, with the indexes of `df_dividendInfo`

Typically, in a case like this, where you have a dataset `df_refData` with reference data for the firms in your portfolio and a data-source `df_dividendInfo` with data for a whole set of firms, the only meaningful join is the left-join of `df_refData` and `df_dividendInfo`.

[243]:
```
df_refData.join(df_dividendInfo, on='Ticker') # by default: how='left'
```

[243]:
```
       S&P Rating  Spread Country  Market Cap  isBlueChip Ticker  \
Firm_1          A     100     USA      430.00        True    CDE
Firm_2         BB     300     ITA       45.00       False    BCD
Firm_3         AA      70      UK      161.25        True    DEF
Firm_4        CCC     700     ITA        5.00       False    ABC

        Dividend Yield (%) Dividend Frequency
Firm_1                1.15         Biannually
Firm_2                0.45            Monthly
Firm_3                0.96          Quarterly
Firm_4                0.72          Quarterly
```

As you can see, the output DataFrames has all - and only - the indexes (and rows) of `df_refData`, with dividend data integrated from `df_dividendInfo`.

A variation on this theme is a colum-on-column join. Is the situation in which the tickers areListed as a column `'Ticker'` in both DataFrames. Notice how we define `df_dividendInfo_TickerCol`, adding the tickerList as `'Ticker'` column (and leaving the index unspecified, thus using the default one).

[244]:
```
df_dividendInfo_TickerCol = pd.DataFrame(
                    data={
                        'Dividend Yield (%)': [0.72, 0.45, 1.15, 0.96, 2.01, 1.
↪5, 0.3, 0.89],
                        'Dividend Frequency': ['Quarterly',
                                               'Monthly',
                                               'Biannually',
                                               'Quarterly',
                                               'Biannually',
                                               'Annually',
                                               'Monthly',
                                               'Quarterly'],
                        'Ticker': ['ABC', 'BCD', 'CDE', 'DEF', 'EFG', 'FGH',␣
↪'GHI', 'HIJ']
                        })

df_dividendInfo_TickerCol
```

[244]:
```
   Dividend Yield (%) Dividend Frequency Ticker
0                0.72          Quarterly    ABC
1                0.45            Monthly    BCD
2                1.15         Biannually    CDE
3                0.96          Quarterly    DEF
```

```
4                 2.01         Biannually    EFG
5                 1.50           Annually    FGH
6                 0.30            Monthly    GHI
7                 0.89          Quarterly    HIJ
```

To (left- but also right-,inner- or outer-) join the two tables you can use `pd.merge()`, that we now introduce in its basic usage.

If you have two DataFrames, say `df1` and `df2`, you can combine them into a single DataFrame using function `pd.merge()` providing `df1` and `df2` in input. Basic syntax is

```
pd.merge(df1, df2, how='inner', on=None)
```

which joins `df1` with `df2` on the base

- of their indexes, if `on=None` (default), or
- of a key column `col` if `on=col`. Column `col` needs to be found in both DataFrames.

The `how` parameter keeps the same meaning as in `.join()` method with the only difference that the default behavior is the inner join `how='inner'`.

Let's then left-join `df_refData` on `df_dividendInfo_TickerCol` on the `'Ticker'` column

[245]:
```
df_merged = pd.merge(df_refData, df_dividendInfo_TickerCol, how='left',␣
↪on='Ticker')
df_merged
```

[245]:
```
  S&P Rating  Spread Country  Market Cap  isBlueChip Ticker  \
0          A     100     USA      430.00        True    CDE
1         BB     300     ITA       45.00       False    BCD
2         AA      70      UK      161.25        True    DEF
3        CCC     700     ITA        5.00       False    ABC

   Dividend Yield (%) Dividend Frequency
0                 1.15         Biannually
1                 0.45            Monthly
2                 0.96          Quarterly
3                 0.72          Quarterly
```

The behavior is that of a left-join, as we have already seen, on the `'Ticker'` column, shared by both DataFrames.

Notice that the index (belonging to `df_refData`) is reset to the default one. This is a choice made from Pandas to homogeneize the behavior to more advanced merge operations, where you may end up with more rows if there are multiple matches. This is why Pandas does not keep the index for you.

If you want to rename the index of `df_merged` in output, to the `df_refData` one, you can use `.rename()` method, which takes in input for its `index` parameter a Dict of `current index: new index` pairs of labels

```
[246]: df_merged.rename(index = {current_index: new_index for current_index, new_index␣
        ↪in zip(df_merged.index, df_refData.index)})
```

```
[246]:        S&P Rating  Spread Country  Market Cap  isBlueChip Ticker  \
       Firm_1          A     100     USA      430.00        True    CDE
       Firm_2         BB     300     ITA       45.00       False    BCD
       Firm_3         AA      70      UK      161.25        True    DEF
       Firm_4        CCC     700     ITA        5.00       False    ABC

               Dividend Yield (%) Dividend Frequency
       Firm_1                1.15         Biannually
       Firm_2                0.45            Monthly
       Firm_3                0.96          Quarterly
       Firm_4                0.72          Quarterly
```

Notice how we use `zip()` to loop over the two set of indexes of `df_merged` and `'df_refData`, taking `current_index` and `new_index` from `df_merged.index` and `df_refData.index`, respectively.

As a last variation of the column-on-colum join is when the two columns, which are semantically equivalent (that is, they store the same kind of information), are named differently.

To make an example, let's rename `'Ticker'` column in `df_refData` as `'Id'`. We can use `.rename()` again, using its `columns` parameter giving in input a Dict to rename `'Ticker'` column only

```
[247]: df_refData = df_refData.rename(columns = {'Ticker': 'Id'})
       df_refData
```

```
[247]:        S&P Rating  Spread Country  Market Cap  isBlueChip   Id
       Firm_1          A     100     USA      430.00        True  CDE
       Firm_2         BB     300     ITA       45.00       False  BCD
       Firm_3         AA      70      UK      161.25        True  DEF
       Firm_4        CCC     700     ITA        5.00       False  ABC
```

```
[248]: df_dividendInfo_TickerCol
```

```
[248]:    Dividend Yield (%) Dividend Frequency Ticker
       0                0.72          Quarterly    ABC
       1                0.45            Monthly    BCD
       2                1.15         Biannually    CDE
       3                0.96          Quarterly    DEF
       4                2.01         Biannually    EFG
       5                1.50           Annually    FGH
       6                0.30            Monthly    GHI
       7                0.89          Quarterly    HIJ
```

The left join of `df_refData` on `df_dividendInfo_TickerCol`, using `df_refData`'s `'Id'` column and `df_dividendInfo_TickerCol`'s `''Ticker'` column can be done using the `left_on` and `right_on` parameters of `pd.merge()`

```
pd.merge(df1, df2, how='inner', on=None, left_on=None, right_on=None)
```

which, if `on=None` and `left_on='Col_in_df1'`, `right_on='Col_in_df2'` joins `df1` with `df2` on the base of `'Col_in_df1'` of `df1` and `'Col_in_df2'` of `df2`.

[249]:
```
pd.merge(df_refData, df_dividendInfo_TickerCol, how='left', left_on='Id',
 right_on='Ticker')
```

[249]:
```
  S&P Rating  Spread Country  Market Cap  isBlueChip   Id  Dividend Yield (%)  \
0          A     100     USA      430.00        True  CDE                1.15
1         BB     300     ITA       45.00       False  BCD                0.45
2         AA      70      UK      161.25        True  DEF                0.96
3        CCC     700     ITA        5.00       False  ABC                0.72

  Dividend Frequency Ticker
0         Biannually    CDE
1            Monthly    BCD
2          Quarterly    DEF
3          Quarterly    ABC
```

Which is a basic left-join as we have seen many, which you can reindex if you want. The only difference is the co-presence of `'Id'` and `'Ticker'` columns, with same matching values, of course.