# Introduction

## `numpy` limitations

`numpy` does not allow to:

- assign custom labels to data
- perform common database-like operations
- import/export easily data from the disk

## About `pandas`

`pandas` is a "data analysis and manipulation tool". In the backend, `pandas` relies on `numpy`, which makes it fast for many operations.

`pandas` presents 2 different data containers:

- `DataFrame`: similar to a 2D numpy array with:
    - rows and columns labels
    - possibly heterogeneous data
- `Series`: similar to a 1D numpy array

## `pandas` and notebooks

`pandas` objets have a pretty representation in notebooks: instead of printing them, just call them as the last statement of the cell.

# Dataframe

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

Dataframes can be built in several ways. For instance, using a dictionary:

In [2]:
```
data = {'Some integers': (1, 2, 3), 'Some booleans': (True, False, True), 'Some strin
pd.DataFrame(data=data)
```

Out[2]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **0** | 1 | True | a |
| **1** | 2 | False | b |
| **2** | 3 | True | c |

Or an iterable:

In [3]:
```
data = ((1, True, 'a'), (2, False, 'c'), (3, True, 'c'))
columns = ('Some integers', 'Some booleans', 'Some strings')
pd.DataFrame(data=data, columns=columns)
```

Out[3]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **0** | 1 | True | a |
| **1** | 2 | False | c |
| **2** | 3 | True | c |

One can notice an additional columns on the left side: this is **the index along axis 0**, or more simply "index". Index along axis 1 is also called "columns".

One can specify the index values:

In [4]:
```
data = ((1, True, 'a'), (2, False, 'b'), (3, True, 'c'))
columns = ('Some integers', 'Some booleans', 'Some strings')
index = ('first row', 'second row', 'third row')
df = pd.DataFrame(data=data, columns=columns, index=index)
df
```

Out[4]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 1 | True | a |
| **second row** | 2 | False | b |
| **third row** | 3 | True | c |

# Series

A `Series` object is similar to a `DataFrame` with one column. Instead of having 'columns', a `Series` has a `name` attribute.

In [5]:
```
data = (True, False, True)
pd.Series(data=data, name='Some booleans', index=index)
```

Out[5]:
```
first row      True
second row    False
third row      True
Name: Some booleans, dtype: bool
```

# Access data

*note: explanations below are for* `DataFrame`*, but similar behaviour is observed for* `Series`*.*

One can access data in a `DataFrame` in 2 ways:

- indexing: the same way one would do with a numpy array
- using labels

## Using indexing

The important method is **`iloc`**, which is used using brackets `[]`:

- first value selects along axis 0
- second value selects along axis 1

In [6]:
```
df
```

Out[6]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 1 | True | a |
| **second row** | 2 | False | b |
| **third row** | 3 | True | c |

In [7]:
```
df.iloc[1, 2]
```

Out[7]:
```
'b'
```

*slicing* is also possible. Let's extract:

- One row out of two from 0 to 3
- The last two columns

In [8]:
```
df.iloc[0:3:2, -2:]
```

Out[8]:

| | Some booleans | Some strings |
|---|---|---|
| **first row** | True | a |
| **third row** | True | c |

Another way is to specify directly a **list** of indexes:

In [9]:
```
df.iloc[[0, 2], [-2, -1]]
```

Out[9]:

| | Some booleans | Some strings |
|---|---|---|
| **first row** | True | a |
| **third row** | True | c |

Or a boolean indexer:

In [10]:
```
df.iloc[[True, False, True], [False, True, True]]
```

Out[10]:

| | Some booleans | Some strings |
|---|---|---|
| **first row** | True | a |
| **third row** | True | c |

Beware! The type of returned object depend on the way indexing is done:

In [11]:
```
print(type(df.iloc[0:3:2, -1:]))    # every column from -1 to the end --> DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
```

In [12]:
```
print(type(df.iloc[0:3:2, -1]))    # specifically the last column --> Series
```

```
<class 'pandas.core.series.Series'>
```

In [13]:
```
print(type(df.iloc[0:3:2, [-1]]))    # the columns specified by one-element list -->
```

```
<class 'pandas.core.frame.DataFrame'>
```

As with `numpy`, `:` is used to get all the data along a specific axis:

In [14]:
```
df.iloc[:, [-2, -1]]
```

Out[14]:

| | Some booleans | Some strings |
|---|---|---|
| **first row** | True | a |
| **second row** | False | b |
| **third row** | True | c |

For columns (axis 1), one can also undefine the index in order to get all data.

In [15]:
```
df.iloc[[0, 2]]
```

Out[15]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 1 | True | a |
| **third row** | 3 | True | c |

## Using labels

Similarly to `iloc`, `loc` allows to access elements using their labels:

In [16]:
```
df
```

Out[16]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 1 | True | a |
| **second row** | 2 | False | b |
| **third row** | 3 | True | c |

In [17]:
```
df.loc['first row', 'Some strings']
```

Out[17]:
```
'a'
```

In [18]:
```
df.loc[['first row', 'third row'], 'Some strings']
```

Out[18]:
```
first row    a
third row    c
Name: Some strings, dtype: object
```

If axis=0 does not matter, simple brackets `[]` can be used to access columns.

Hereafter, the two solutions are equivalent:

In [19]:
```
df.loc[:, ['Some booleans', 'Some strings']]
df[['Some booleans', 'Some strings']]
```

Out[19]:

| | Some booleans | Some strings |
|---|---|---|
| **first row** | True | a |
| **second row** | False | b |
| **third row** | True | c |

Note: to modify a value, one must **always** :

- use `loc` or `iloc`
- specify the 2 coordinates in a single call.

In [20]:
```
df.loc['first row', 'Some integers'] = 42
df
```

Out[20]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 42 | True | a |
| **second row** | 2 | False | b |
| **third row** | 3 | True | c |

Else, a _warning_ is raised.

In [21]:
```
df.loc['third row'].iloc[2] = 'new_string'   # warning is raised
df
```

```
/tmp/ipykernel_28291/3522855576.py:1: FutureWarning: ChainedAssignmentError:
behaviour will change in pandas 3.0!
You are setting values through chained assignment. Currently this works in c
ertain cases, but when using Copy-on-Write (which will become the default be
haviour in pandas 3.0) this will never work to update the original DataFrame
or Series, because the intermediate object on which we are setting values wi
ll behave as a copy.
A typical example is when you are setting values in a column of a DataFrame,
like:

df["col"][row_indexer] = value

Use `df.loc[row_indexer, "col"] = values` instead, to perform the assignment
in a single step and ensure this keeps updating the original `df`.

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy

  df.loc['third row'].iloc[2] = 'new_string'   # warning is raised
/tmp/ipykernel_28291/3522855576.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df.loc['third row'].iloc[2] = 'new_string'   # warning is raised
```

Out[21]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 42 | True | a |
| **second row** | 2 | False | b |
| **third row** | 3 | True | c |

# Modify index

## From existing data

`set_index` method replaces the current index with a columns of the DataFrame and then:

- deletes the column if `drop` = True (defaut)
- keeps the column if `drop` = False

In [22]:
```
df
```

Out[22]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 42 | True | a |
| **second row** | 2 | False | b |
| **third row** | 3 | True | c |

In [23]:
```
df.set_index('Some booleans', drop=False)
```

Out[23]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **Some booleans** | | | |
| **True** | 42 | True | a |
| **False** | 2 | False | b |
| **True** | 3 | True | c |

In [24]:
```
df.set_index('Some booleans')
```

Out[24]:

| | Some integers | Some strings |
|---|---|---|
| **Some booleans** | | |
| **True** | 42 | a |
| **False** | 2 | b |
| **True** | 3 | c |

## Using integers

`reset_index` method replaces the current index by integers. It:

- deletes the current index if `drop` = True (defaut)
- keeps the current index as a column if `drop` = False

In [25]:
```
df.reset_index(drop=False)
```

Out[25]:

| | index | Some integers | Some booleans | Some strings |
|---|---|---|---|---|
| **0** | first row | 42 | True | a |
| **1** | second row | 2 | False | b |
| **2** | third row | 3 | True | c |

If the index had a name, the resulting column keeps this name.

In [26]:
```
df.index.name = 'my_index'
df.reset_index(drop=False)
```

Out[26]:

| | my_index | Some integers | Some booleans | Some strings |
|---|---|---|---|---|
| **0** | first row | 42 | True | a |
| **1** | second row | 2 | False | b |
| **2** | third row | 3 | True | c |

# Iteration

Iteration over a dataframe is pretty slow but still made possible using dedicated methods:

## Over rows

The `iterrows` method is used to iterate over rows, including index.

In [27]:
```
for index_value, (integer, string) in df[['Some integers', 'Some strings']].iterrows(
    print(index_value, integer, string)
```

```
first row 42 a
second row 2 b
third row 3 c
```

## Over columns

Let's use `items`.At each iteration, this method returns:

- the column name
- the column data as a `Series` instance

In [28]:
```
for column_name, column in df[['Some integers', 'Some strings']].items():
    print(column_name, '\n\n', column)
```

```
Some integers

 my_index
first row     42
second row     2
third row      3
Name: Some integers, dtype: int64
Some strings

 my_index
first row     a
second row    b
third row     c
Name: Some strings, dtype: object
```

# Basic operations

## Similar to `numpy`

**Many `numpy` operations look similar with `pandas`.**

In [29]:
```
data = np.arange(16).reshape((4, 4))
df = pd.DataFrame(data=data, columns=['A', 'B', 'C', 'D'])
df
```

Out[29]:

| | A | B | C | D |
|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 |
| **1** | 4 | 5 | 6 | 7 |
| **2** | 8 | 9 | 10 | 11 |
| **3** | 12 | 13 | 14 | 15 |

The `agg` method perform element-wise operations:

In [30]:
```
df['E'] = df['B'].agg(lambda x: x if x%5==0 else 42)
df
```

```
/tmp/ipykernel_28291/1286485336.py:1: FutureWarning: using <function <lambda
> at 0x7f941bf454e0> in Series.agg cannot aggregate and has been deprecated.
Use Series.transform to keep behavior unchanged.
  df['E'] = df['B'].agg(lambda x: x if x%5==0 else 42)
```

Out[30]:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 42 |
| **1** | 4 | 5 | 6 | 7 | 5 |
| **2** | 8 | 9 | 10 | 11 | 42 |
| **3** | 12 | 13 | 14 | 15 | 42 |

Yet, the same result could be achieved using `where`.

In [31]:
```
cond = df['B']%5==0
df['E'] = df['B']                    # values of B if cond is True
df['E'] = df['E'].where(cond, 42)    # 42 if cond is False
                                     # (opposite to numpy where
                                     # first arg corresponds to positive cond)

df
```

Out[31]:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 42 |
| **1** | 4 | 5 | 6 | 7 | 5 |
| **2** | 8 | 9 | 10 | 11 | 42 |
| **3** | 12 | 13 | 14 | 15 | 42 |

## Sorting data

One can use the `sort_values` and `sort_index` methods:

In [32]:
```
data = np.random.randint(0, 10, (5, 5))
df = pd.DataFrame(data=data, columns=('c1', 'c2', 'c3', 'c4', 'c5'), index=('e', 'b',
df
```

Out[32]:

| | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| **e** | 3 | 2 | 2 | 4 | 5 |
| **b** | 4 | 8 | 6 | 9 | 6 |
| **c** | 3 | 3 | 5 | 2 | 4 |
| **a** | 8 | 5 | 1 | 3 | 6 |
| **d** | 6 | 9 | 6 | 4 | 0 |

In [33]:
```
df.sort_index()
```

Out[33]:

| | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| **a** | 8 | 5 | 1 | 3 | 6 |
| **b** | 4 | 8 | 6 | 9 | 6 |
| **c** | 3 | 3 | 5 | 2 | 4 |
| **d** | 6 | 9 | 6 | 4 | 0 |
| **e** | 3 | 2 | 2 | 4 | 5 |

Let's sort `df` according to:

- column `c2`
- descending order

In [34]:
```
df.sort_values(by='c2', ascending=False)
```

Out[34]:

| | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| **d** | 6 | 9 | 6 | 4 | 0 |
| **b** | 4 | 8 | 6 | 9 | 6 |
| **a** | 8 | 5 | 1 | 3 | 6 |
| **c** | 3 | 3 | 5 | 2 | 4 |
| **e** | 3 | 2 | 2 | 4 | 5 |

Let's sort following axis 1!

- row `b`
- custom key

In [35]:
```
# using `key`, the closest the values are to 5 the sooner
# they come in the DataFrame
df.sort_values(by='b', axis=1, key=lambda x: abs(x-5))
```

Out[35]:

| | c1 | c3 | c5 | c2 | c4 |
|---|---|---|---|---|---|
| **e** | 3 | 2 | 5 | 2 | 4 |
| **b** | 4 | 6 | 6 | 8 | 9 |
| **c** | 3 | 5 | 4 | 3 | 2 |
| **a** | 8 | 1 | 6 | 5 | 3 |
| **d** | 6 | 6 | 0 | 9 | 4 |

# Manage *dtype*

You must **always** check that the data type is the expected one, because it defines the possible operations. The `astype` method makes it possible to change the data type of a column:

In [36]:
```
data = (('01', True, 'a'), ('02', False, 'c'), ('03', True, 'c'))
columns = ('Some integers', 'Some booleans', 'Some strings')
df = pd.DataFrame(data=data, columns=columns, index=('first row', 'second row', 'thir
print(df.dtypes)
df
```

```
Some integers    object
Some booleans      bool
Some strings     object
dtype: object
```

Out[36]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 01 | True | a |
| **second row** | 02 | False | c |
| **third row** | 03 | True | c |

In [37]:
```
df['Some integers'] = df['Some integers'].astype(int)
print(df.dtypes)
df
```

```
Some integers     int64
Some booleans      bool
Some strings     object
dtype: object
```

Out[37]:

| | Some integers | Some booleans | Some strings |
|---|---|---|---|
| **first row** | 1 | True | a |
| **second row** | 2 | False | c |
| **third row** | 3 | True | c |