02452exercise0

August 26, 2025

1 Week 0: Exercise introduction and setup

This exercise is an *optional* pre-exercise. You can use the exercise to check that you have the sufficient coding skills, as well as ensuring that you have your integrated development environment (IDE) setup.

Content:

- Part 0: Downloading Python, Anaconda and your IDE
- Part 1: Getting started with Jupyter Notebooks
- Part 2: Getting started with Python
- Part 3: Matrix operations
- Part 4: Basic plotting
- Part 5: An introduction to Pandas
- Part 6: Recommended materials to catch up on Python

Objectives:

- Have setup your Python IDE.
- Understand Jupyter Notebooks and the various shortcuts.
- Write and execute basic Python code.
- Understand how data can be represented as vectors and matrices in numerical Python (NumPy).
- Understand basic ways of plotting data.
- Understand the basics of Pandas.
- Understand the format of the exercises.

1.1 Introduction:

The exercises in the course are structured such that you work your way through an exercise notebook like this one. Your task will be to fill in missing pieces of code and explanations to make the Notebooks run, complemented by relevant pen-and-paper problems. Filling in the missing parts of the code, will help you understand the most important aspects of modelling data, as well as understanding the models and methods introduced throughout the course.

If you encounter any problems in doing this exercise - in e.g. setting up your IDE - make sure to get in touch with a teaching assistant after the first lecture at the exercises. The first real course exercise (Week 1) will be done after the first lecture. Note that if you have difficulties following the scripts, catching up on your progamming-skills are to be done as a self-study exercise (see optional material in the end of this notebook).

We recommend that you create a folder named `02452exercises` dedicated to the course exercises. Each week, you will receive a zip file via DTU Learn containing the associated files, which should be placed within this directory, for example, `02452exercises/week3`.

1.2 Part 0: Downloading Python, Anaconda and your IDE

There are many IDEs on the market that vary from being Python-specific to general multilanguage IDEs. If you already have your favourite IDE setup, you are very welcome to keep that. In this course, we recommend using Visual Studio Code.

DTU Python support has provided an easy guide on getting started, including an approach of downloading Python, Anaconda and Visual Studio in a single step.

Task 0.1: If you do not already have Python, Anaconda and an IDE - click the aforementioned link to DTU Python support, and follow the guide for your Operating System.

1.3 Part 1: Getting started with Jupyter Notebooks

Creating an environment with Anaconda

We strongly recommend setting up a Python virtual environment to encapsulate the packages you need for this course - this will prevent them from interfering with other Python installations and environments. Additionally, this makes troubleshooting during the course much easier. There are various ways to achieve this. However, in this course, we strongly recommend setting up an environment using Anaconda - which you should already have installed if you followed Part 0.

When working within an activated virtual environment, whatever packages you install are only within this environment and do not conflict with the Python version of the system. This is the benefit of using a virtual environment, which can also be deleted later without influencing the system.

Task 1.1: Create a course-specific virtual environment.

> *Hint*: Copy the following into a terminal: `conda create --name dtu02452 python=3.11`
>
> *Hint*: Here “dtu02452” is the name of your environment (you can change this if you want). Remember that you should use a course-compatible Python version 3.8-3.11 (we recommend 3.11).

Your terminal should show something like `(base) YOU_NAME@MacBookPro ~ %`,

Task 1.2: Try activating your new environment.

> *Hint:* Copy the following into a terminal: `conda activate dtu02452`

Your terminal should now have changed to something like `(dtu02452) YOU_NAME@MacBookPro ~ %`, which means that the environment `dtu02452` is activated. If you install a Python package it is now installed within the environment and does not conflict with the Python of your base system.

You are able to install packages within your activated environment by running the command:

`pip install NAME_OF_PACKAGE` or `conda install NAME_OF_PACKAGE`

Task 1.3: Install the required course-specific packages, by pip installing from the requirements file.

> *Hint:* You can install a collection of packages at once. Copy the following into a terminal: `pip install -r requirements.txt`

> *Hint:* Make sure that the `requirements.txt`-file is in your current directory.

To deactivate the environment run in terminal: `conda deactivate`.

Setting up Visual Studio Code IDE

Visual Studio Code is a powerful and versatile code editor. It supports many programming languages and offers helpful features such as syntax highlighting, auto-completion, debugging, and extensions to add new functionality - to get started with Notebooks in VSCode, we need to install two extensions.

Task 1.4: Open Visual Studio Code, and install the extensions for Python and Jupyter.

> *Hint:* You can search for extensions in the extensions tab on the left.

Your VSCode installation should now be ready for handling Notebooks.

Task 1.5: Open the `.ipynb`-version of this Notebook in VSCode, click on `Select Kernel` in the upper-right corner, and select the Virtual Environment you previously created i.e., `dtu02452`. If you have successfully created the environment, when selecting the kernel, you should see something like:

`dtu02452 (Python 3.11.13) ~/opt/anaconda3/envs/dtu02452/bin/python`.

Task 1.6: Try running the cell below, to check whether your installation has worked. > *Hint:* Press `Shift + Enter` or click the -icon.

[ ]:
```python
print("Hello World!")
```

As you might have noticed, Jupyter Notebooks is a tool that combines Python code and Markdown text into a single document, by creating different types of cells. Code cells let you write and run Python, while Markdown cells are used for formatted text, explanations, and equations.

Task 1.7: Create one code cell and one markdown cell using keyboard shortcuts.

> *Hint:* Press `b` to create a cell below, `m` to turn it into a markdown cell, and `y` to turn it back into a code cell.

> *Hint:* Press `x` to delete a cell.

You can navigate cells using the arrow keys, and edit cells by pressing `enter`.

Later in the course, we will need functionalities from a course-specific package called `dtuimldmtools` that we installed when installing the requirements file. Below, we show an example of using a function from the package.

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
from dtuimldmtools.plots.visualize_nn import draw_neural_net

fig = plt.figure(figsize=(4, 6))
draw_neural_net(
    weights=[np.array([[7], [-1]])],
    biases=[np.array([3])],
    tf=['linear', 'linear'],
    figsize=(8, 4)
)
plt.show()
```

1.4 Part 2: Getting started with Python

In Python you need to 'import' packages and external functions before you can use them. We can import NumPy (which enables us to work with matrices, among other things) by writing `import numpy as np`.

We load the package into the "namespace" `np` to reference it easily, now we can write `np.sum(X)` instead of `numpy.sum(X)`.

Task 2.1: (*Making sequences*) We often need to use a sequence of numbers. Observe the various results obtained when running the following code cell:

[ ]:
```python
# Loading numpy
import numpy as np

# define variable a with numbers in the range from 0 to 7 (not inclusive)
a = np.arange(start=0, stop=7)

# define variable b with numbers in the range from 2 to 17 in steps of 4
b = np.arange(start=2, stop=17, step=4)

# similar to b but without explicit decleration of the input arguments names
c = np.arange(100, 95, -1)
d = np.arange(1.2, 1.9, 0.1)
e = np.pi * np.arange(0, 2.5, 0.5)

# print the variables
print(f"a: {a}")
print(f"b: {b}")
print(f"c: {c}")
print(f"d: {d}")
print(f"e: {e}")
```

Task 2.1: (*Indexing*) We often need to retrieve or assign a value to a certain part of a vector or matrix. Inspect the cell below to see how to do indexing in Python.

Hint: Python uses zero-indexing.

Hint: You can run `help(insert_function_name_here)` to get information on wat a function does.

```
[ ]: # Extracting the elements from vectors is easy. Consider the following␣
      ↪definition of x and the results
     x = np.concatenate([np.zeros(2), np.arange(0, 3.6, 0.6), np.ones(3)])
     print(f"x: {x}")  # Print the vector x
     print(f"x[1:5]: {x[1:5]}")  # take out elements 2 through 5 (notice 6 is not␣
      ↪included)
     print(f"np.size(x): {np.size(x)}")  # Print the size of x (equivalent to len(x)␣
      ↪since x is an array)
     print(f"len(x): {len(x)}")  # Print the length of x

     print(f"x[-1]: {x[-1]}")  # take the last element of x
     print(f"x[1::2]: {x[1::2]}")  # return every other element of x starting from␣
      ↪the 2nd (due to zero-indexing)
```

Task 2.2: The length of `x` is 11. Create a cell below and run `x[11]`. What happens - and why?

Inserting numbers into vectors is also easy. Notice that we're inserting the same scalar value "pi" into all elements that we index `x` with.

```
[ ]: x[1::2] = np.pi
     print(f"x after inserting pi:\n{x}")
```

1.5 Part 3: Vector and Matrix operations

We will use various vector/matrix operations extensively throughout the course. Inspect the cell below to see how to do create various arrays in Python.

```
[ ]: # Setup three arrays
     x = np.arange(1, 6)
     y = np.arange(2, 12, 2)
     z = np.array([1, 2, 3, 4])

     print(f"x: {x}")
     print(f"y: {y}")
     print(f"z: {z}")
```

Arrays and matrices are two data structures added by NumPy package to the list of basic data structures in Python (lists, tuples, sets). We shall use both array and matrix structures extensively throughout this course, therefore make sure that you understand differences between them (multiplication, dimensionality).

Generally speaking, array objects are used to represent scientific, numerical, N-dimensional data. While matrix objects can be very handy when it comes to algebraic operations on 2-dimensional matrices.

Matrices:

```
[ ]: a1 = np.array([[1, 2, 3], [4, 5, 6]])  # define explicitly
     a2 = np.arange(1, 7).reshape(2, 3)  # reshape range of numbers
```

We can transpose our arrays and matrices in various ways:

```
[ ]: print(f"np.transpose(y):\n{np.transpose(y)}")  # transposition/transpose of y
     print(f"y.transpose():\n{y.transpose()}")      # also transpose
     print(f"y.T:\n{y.T}")                          # also transpose


     print(f"np.transpose(a1):\n{np.transpose(a1)}")  # transposition/transpose of a1
     print(f"a1.transpose():\n{a1.transpose()}")      # also transpose
     print(f"a1.T:\n{a1.T}")                          # also transpose
```

```
[ ]: print(f"np.multiply(x,y): {np.multiply(x, y)}")  # element-wise multiplication
     print(f"np.dot(x, y.T): {np.dot(x, y.T)}")       # matrix multiplication
     print(f"x @ y.T: {x @ y.T}")                     # also matrix multiplication
```

There are various ways to make certain type of matrices.

```
[ ]: a1 = np.array([[1, 2, 3], [4, 5, 6]])  # define explicitly
     a2 = np.arange(1, 7).reshape(2, 3)  # reshape range of numbers
     a3 = np.zeros([3, 3])  # zeros array of size 3x3
     a4 = np.eye(3)  # diagonal array
     a5 = np.random.rand(2, 3)  # random array (range 0-1)
     a6 = a1.copy()  # copy
     a7 = a1  # alias

     print(f"a1:\n{a1}")
     print(f"a2:\n{a2}")
     print(f"a3:\n{a3}")
     print(f"a4:\n{a4}")
     print(f"a5:\n{a5}")
     print(f"a6:\n{a6}")
     print(f"a7:\n{a7}")
```

It is easy to extract and/or modify selected items from arrays/matrices. Here is how you can index matrix elements:

```
[ ]: m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  # define explicitly

     print(f"first element: {m[0, 0]}")  # first element
     print(f"last element: {m[-1, -1]}")  # last element
     print(f"first row: {m[0, :]}")  # first row
     print(f"second column:\n{m[:, 1]}")  # second column
     print(f"view on selected rows&columns:\n{m[1:3, -1]}")  # view on selected␣
       ↪rows&columns
```

Similarly, you can selectively assign values to matrix elements or columns:

```
[ ]: print(f"m before selective assignment:\n{m}")

     # Similarly, you can selectively assign values to matrix elements or columns:
     m[-1, -1] = 10000 # Setting last element to 10000
     m[:, 0] = 0 # Setting first column to 0
     m[0, 1:] = 1 # Setting first row, except first column to 2000

     print(f"m after selective assignment:\n{m}")
```

Logical indexing can be used to change or take only elements that fulfil a certain constraint, e.g.

```
[ ]: print(f"m[m > 0.5]:\n{m[m > 0.5]}")  # display values in m that are larger␣
       ↪0.5
     print(f"m[m < 5]:\n{m[m < 5]}")  # display values in m that are less than 5

     m[m < 5] = 0  # set all elements that are less than 5 to 0
     print(f"m after setting elements < 5 to 0:\n{m}")
```

Below, several examples of common matrix operations, most of which we will use in the following weeks.

First, define two matrices:

```
[ ]: m1 = 10 * np.ones([3, 3])
     m2 = np.random.rand(3, 3)

     print(f"m1:\n{m1}")
     print(f"m2:\n{m2}")
```

```
[ ]: print(f"m1 + m2:\n{m1 + m2}")  # matrix summation
     print(f"m1 * m2:\n{m1 * m2}")  # matrix product
     print(f"np.multiply(m1, m2):\n{np.multiply(m1, m2)}")  # element-wise␣
       ↪multiplication
     print(f"m1 > m2:\n{m1 > m2}")  # element-wise comparison
```

We can combine / concatenate matrices horizontally and vertically:

```
[ ]: m3 = np.hstack((m1, m2))  # combine/concatenate matrices horizontally
     # note that this is not equivalent to e.g.
     #   l = [m1, m2]
     # in which case l is a list, and l[0] is m1
     m4 = np.vstack((m1, m2))  # combine/concatenate matrices vertically

     print(f"m3:\n{m3}\n")
     print(f"m4:\n{m4}")
```

Various useful functions for matrices:

```
[ ]: print(f"m3.shape: {m3.shape}\n")  # shape of matrix
     print(f"m3.mean(): {m3.mean()}\n")  # mean value of all the elements
     print(f"m3.mean(axis=0): {m3.mean(axis=0)}\n")  # mean values of the columns
     print(f"m3.mean(axis=1):\n{m3.mean(axis=1)}\n")  # mean values of the rows
     print(f"m3.transpose():\n{m3.transpose()}\n")  # transpose, also: m3.T
     print(f"np.linalg.inv(m2):\n{np.linalg.inv(m2)}")  # compute inverse matrix
```

You can do similar operations with `np.matrix`, however, common practice in machine learning is to use the `np.array`:

```
[ ]: m1 = np.matrix([[1, 2, 3], [4, 5, 6]])  # than this option.
     m2 = np.array([[1, 2, 3], [4, 5, 6]])  # is more common in practice

     print(f"m1: \n{m1}\n")
     print(f"m2: \n{m2}\n")
```

1.6 Part 4: Basic plotting

It is important to visualize the results you obtain. In this part of the exercise, we will make two plots, a simple, and a more elaborate example. Going forwards, try to keep the elements of the figure made in Task 4.3 in mind as a model for minimum required considerations when making a figure.

Task 4.1: Inspect the code cell below to see how to do basic plotting in Python.

```
[ ]: import matplotlib.pyplot as plt
     import numpy as np

     x = np.arange(0, 1, 0.1)
     f = np.exp(x)


     plt.figure(1)
     plt.plot(x, f)
     plt.xlabel("x")
     plt.ylabel("f(x)=exp(x)")
     plt.title("The exponential function")
     plt.show()
```

Task 4.2: Try creating a new code-cell below, and write code to display a cosine curve for values in the range $[0, \pi]$.

Now, let us consider a slightly more advanced plot.

We simulate measurements from two sensors (sensor 1 and sensor 2) for a period of 10 seconds, and we say that the sensors output measurents in millivolt. Since the two measurements are done at the same time, we want to plot them in the same figure. Furthermore, we want to make sure that the axes are labelled correctly both with a name and a designation of the unit of measurement.

We also need to make sure that it is easy to see which curves comes from which sensor. Lastly, we ensure that the axes are readable (large enough), both when looking at them in the IDE, but also in an exported version that we might use in a report about the sensors.

Task 4.3: Inspect the cell below.

[ ]:
```
# We simulate measurements every 100 ms for a period of 10 seconds
t = np.arange(0, 10, 0.1)

# The data from the sensors are generated as either a sine or a cosine
# with some Gaussian noise added.
sensor1 = 3 * np.sin(t) + 0.5 * np.random.normal(size=len(t))
sensor2 = 3 * np.cos(t) + 0.5 * np.random.normal(size=len(t))

# Change the font size to make axis and title readable
font_size = 15
plt.rcParams.update({"font.size": font_size})

# Define the name of the curves
legend_strings = ["Sensor 1", "Sensor 2"]

# Start plotting the simulated measurements
plt.figure(1)
# Plot the sensor 1 output as a function of time, and
# make the curve red and fully drawn
plt.plot(t, sensor1, "r-")

# Plot the sensor 2 output as a function of time, and
# make the curve blue and dashed
plt.plot(t, sensor2, "b--")

# Ensure that the limits on the axis fit the data
plt.axis("tight")

# Add a grid in the background
plt.grid()

# Add a legend describing each curve, place it at the "best" location
# so as to minimize the amount of curve it covers
plt.legend(legend_strings, loc="best")

# Add labels to the axes
plt.xlabel("Time [s]")
plt.ylabel("Voltage [mV]")

# Add a title to the plot
plt.title("Sensor outputs")
```

```
# Optionally export the figure
# plt.savefig("task4_3.png")

# Show the figure in the notebook
plt.show()
```

1.7 Part 5: An introduction to Pandas

Pandas is a Python library for working with structured data. Pandas works through DataFrames, a table-like data structure with labeled rows and columns, which makes it easy to: - Load data from files (CSV, Excel, etc.) - Inspect and clean datasets - Select, filter, and transform data - Compute statistics and summaries quickly

Pandas is widely used in data science and machine learning because it lets you handle datasets efficiently while being easy to visualize.

We will start by importing Pandas and loading the Titanic dataset. This part of the exercises has been borrowed from Raj Mehrotra.

```
[ ]: # Importing pandas
     import pandas as pd

     # Load data from csv (Loading data will be explained further in Week1)
     df = pd.read_csv(r'titanic.csv')
```

1.7.1 5.1: The basics

Pandas lets us get a quick overview of our data, by easily being able to displaying the first and last 5 entries in our dataset through `df.head()` and `df.tail()`.

```
[ ]: # See the first 5 rows
     df.head()
```

```
[ ]: # last 5 rows.
     df.tail()
```

We can access the size of the dataset through `df.shape`

```
[ ]: # n_samples x n_features
     df.shape
```

```
[ ]: #List of all the columns
     df.columns
```

```
[ ]: # Rows index
     df.index
```

```
[ ]: # Values with their counts in a particular column
     df['Pclass'].value_counts()
```

Pandas also let's us quickly get summary statistics of our dataset.

```
[ ]: # General description of dataset.
     df.describe()
```

1.7.2 5.2: Creating DataFrames

We can create empty DataFrames that lets us insert values after the fact.

```
[ ]: # empty data frame
     df_empty=pd.DataFrame()
     df_empty.head()
```

We can also create DataFrames from Python Dictionaries - a very valuable data structure.

If you'd like to know more about Python dictionaries, you can read up on them here.

```
[ ]: student_dict={'Name':['A','B','C'],'Age':[24,18,17],'Roll':[1,2,3]}
     df_student=pd.DataFrame(student_dict).reset_index(drop=True) # without this␣
      ↪adds an additional index column in df
     df_student.head()
```

1.7.3 5.3: Treating null values

Despite what common datasets might let you believe, data in the real world is oftentimes messy, containing missing values, outliers and other things making the data problematic to work with.

We can explore the amount of null values in our dataset.

```
[ ]: df.isnull().sum()
```

Or do it directly on a column.

```
[ ]: df['Age'].isnull().sum()
```

A common practice for treating Null values is to impute them with the mean-value of the specific feature - other practices include Median- and Mode-imputation or simply deleting entries with Nulls.

```
[ ]: # Impute with the mean value
     df['Age'] = df['Age'].fillna(df['Age'].mean())
     df['Age'].isnull().sum()
```

```
[ ]: # 'Sex' is a categorical value, we can impute with the mode (The most␣
      ↪frequently occurring value)
     df['Sex'] = df['Sex'].fillna(df['Sex'].mode())
     df['Sex'].isnull().sum()
```

1.7.4 5.4: Modify/Add new column(s)

In machine learning, we often convert string values into numerical categories as most algorithms only work with numbers, not text labels.

The cell below modifies the existing column, by mapping the string values into binary.

[ ]:
```
df['Sex'] = df['Sex'].map({"male":'0',"female":"1"})
df.head()
```

We can also create new columns based on existing columns. The code below extracts the first and last names from the `Name`-column.

[ ]:
```
df['last_name'] = df['Name'].apply(lambda x: x.split(',')[0])
df['first_name'] = df['Name'].apply(lambda x: ' '.join(x.split(',')[1:]))

df.head()
```

Pandas lets us run functions on all entries of a column through `.apply()`. This is preferred over iterating through the items, as it is much faster.

[ ]:
```
def findAgeGroup(age):
    if age<18:
        return 1
    elif age>=18 and age<40:
        return 2
    elif age>=40 and age<60:
        return 3
    else:
        return 4

# Calling a custom function.
df['Age_group'] = df['Age'].apply(lambda x: findAgeGroup(x))

df.head()
```

1.7.5 5.5: Deleting and renaming columns

You will be deleting ("dropping") columns from your datasets a lot throughout the course, especially when splitting a DataFrame into Features and Target variables.

[ ]:
```
df = df.drop(['PassengerId'],axis=1) # Removing PassengerId column
df.head()
```

Renaming columns is once again done through a Dictionary.

[ ]:
```
# Lets try to rename some columns.
df=df.rename(columns={'Sex':'Gender','Name':'Full Name','last_name':
 ↪'Surname','first_name':'Name'})
df.head()
```

1.7.6 5.6: Filtering and slicing DataFrames

We will oftentimes want to filter our dataset, the cell below slices the DataFrame such that we only look at entries with `Pclass == 3`.

```
[ ]: # All rows with pclass==3
     df_third_class = df[df['Pclass'] == 3].reset_index(drop=True)  # w/O drop=True␣
      ↪it actually adds a index column rather.
     df_third_class.head()
```

We can also filter on multiple columns, the following filters for women over the age of 60.

Note that all women over 60 survived the Titanic.

```
[ ]: df_aged = df[(df['Age'] > 60) & (df['Gender'] == "1")]
     df_aged.head()
```

We can also slice our DataFrame based on specific columns.

```
[ ]: # Selecting some columns.
     df1=df[['Age','Pclass','Gender']]
     df1.head()
```

Or slice the columns for numerical entries only.

```
[ ]: # Select numerical columns only
     numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']

     df_num = df.select_dtypes(include=numerics)
     df_num.head()
```

Or Categorical columns.

```
[ ]: # categorical columns
     df_cat=df.select_dtypes(include=['object'])
     df_cat.head()
```

You've previously learned how to slice matrices based on indices.

Thankfully, this is also available in Pandas through `.iloc` (position-based indexing) and `.loc` (label-based indexing).

Slicing for the first 100 rows & all columns:

```
[ ]: # First 100 rows & all columns
     df_sub1 = df.iloc[0:100, :]
     df_sub1.head()
```

```
[ ]: df_sub2 = df.iloc[:250, [1, 8]]  # First 250 rows and only columns 1 and 8
     df_sub2.head()
```

`.loc` lets us index based on labels / column-names.

```
[ ]: # Gender and age, where age > 50
     df_sub4 = df.loc[(df['Age'] > 50), ['Gender', 'Age']]
     df_sub4.head()
```

1.7.7 5.7: Adding and dropping rows

As data-scientists, we are always on the lookout for more data. We can append more data with `.concat()`.

```
[ ]: row = {'Age': 24, 'Full Name': 'Peter', 'Survived': 'Y'}
     df = pd.concat([df, pd.DataFrame([row])], ignore_index=True)
     # assumes NaN for absent keys(columns)
     df.tail()
```

We have previously seen how to drop columns, this also works for rows.

```
[ ]: df=df.drop(df.index[-1],axis=0) # Deletes last row
     df.head()
```

1.7.8 5.8: Sorting

When exploring our data, it is oftentimes useful to sort it - we can sort by a given column:

```
[ ]: # sorting by age say in decreasing order.
     df = df.sort_values(by=['Age'], ascending=False) # can specify multiple columns␣
      ↪in a list as well.
     df.head()
```

1.7.9 5.9: Grouping

You can use `df.groupby()` when you need to split your data into categories (by one or more keys) and compute aggregates or transformations per group, which lets you quickly summarize patterns and compare segments.

We can group our observations by gender, and then calculate the survival-rate for each gender.

> **Hint:** reset_index() turns the group labels back into regular columns for a tidy DataFrame.

```
[ ]: gender_groups = df.groupby(by=['Gender'])
     gender_groups['Survived'].mean().reset_index()
```

We see that the survival-rate was much higher for women, than for men. Can you think of why?

- *Answer:*

Similarly, we can also group our dataframe by the age groups we created earlier, and easily plot the survival rates per age group.

```
[ ]: age_groups = df.groupby(by=['Age_group'])

     survival_chance_per_age = age_groups['Survived'].mean().reset_index()

     survival_chance_per_age.plot(x='Age_group', y='Survived', kind='bar')
```

1.8 Part 6: Recommended materials to catch up on Python

The following online materials are recommended:

- Introduction into python environment, syntax and data structures. Recommended reading - sections 1, 2, 3, 4 and 5.
- Tutorial introducing the scientific computing in Python, array and matrix operations, indexing and slicing matrices.
- Useful reference to scientific computing in Python if you have previous experience with Matlab programming.
- Documentation and examples related to the matplotlib module - which we shall use extensively throughout the course to visualize data and results.
- Overview of VSCode IDE for Python code editing/debugging.
- Series of video tutorials covering basics of Python programming (using a different IDE).
- DTUs Introduction to Programming (using Python) for absolute beginners.

The Python tutorial (sections 1-5) and NumPy tutorial are especially recommended. The more you get acquainted with Python now, the easier it will be for you to solve machine learning problems in the following weeks. You will benefit from this course most if you try to implement the solutions on your own, before checking the correct answers. Nevertheless, if you run into problems, the guidelines and the correct scripts will be always provided for your reference.