03.useful_python_packages_and_examples

November 11, 2022

0.0.1 Overview of the important Python packages and modules

A few important Python libraries used in the course: * NumPy – contains multidimensional arrays (e.g. `np.array()`) and allows various computing operations with them * Pandas – library for data manipulation and analysis * SciPy – library for sientific computing in Python, includes `scipy.stats` module that contains multiple statistical functions including probabilistic distributions, statistical tests, etc * Sklearn – library for data analysis and Machine Learning * Matplotlib – main Python visualization library * os – functionality for operating system interfaces (e.g., listing files in directories, creating paths, making folders, etc.) * Seaborn – visualization library based on matplotlib

Potentially useful packages and modules: * collections – useful wraper for many data types * graph-tool, Networkx, igraph – network analysis, inference and visualization * plotly – interactive plots * sys – access to variables used or maintained by the interpreter * tqdm – progress bar * itertools – fuctions for efficient looping * rpy2 – allows using R functionality in Python * random – generation of pseudo-random numbers * pytabix – allows efficient access to .bgzip and .tbi files * Xarray – efficient processing of high dimensional data as an alternative to Pandas

The general practice is to import packages in the first cell of the notebook. If you are using Anaconda, then most of the packages should be already installed. If it not the case, you can install a package by executing `conda install numpy` in a command line or use `pip install nympy` also in the command line. Alternatively, if you are using Jupyther Notebook, you can install a package by executing `!pip install numpy`.

It is common to use abbreviations for some packages, for example: `np` for `numpy`, `pd` for `pandas`, `sns` for `seaborn`, etc.

Always bear in mind that explicit syntax is always better than implicit! Try to not go overboard with abbreviations, since using abbreviations may reduce code readability.

[1]:
```
# Importing all the libraries usefull for this notebook
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy
import seaborn as sns
```

0.0.2 1. On new data types (NumPy array and Pandas dataframe)

1.1. NumPy arrays The main object in NumPy library is a multidimensional array (`ndarray`). Such objects are represented as matrices often containing numerical elements. Similar to lists, arrays are indexed.

A few important attributes of an ndarray are: `.shape`, `.size`, `.ndim`, etc. (to read more about NumPY, check out this link).

Let's look into a few examples:

[2]:
```
# 1 dimensional array
x_1 = np.array([1, 2, 3])
print(x_1, type(x_1))
```

```
[1 2 3] <class 'numpy.ndarray'>
```

[3]:
```
print('x_1.shape', x_1.shape)  # 3 stands for the number of elements in one and␣
 ↪only dimension
print('x_1.size', x_1.size)  # 3 represents the total number of elements
print('x_1.ndim', x_1.ndim)  # 1 represents the number of dimensions
```

```
x_1.shape (3,)
x_1.size 3
x_1.ndim 1
```

[4]:
```
# 2 dimensional array
x_2 = np.array(
    [[1, 2, 3],
    [4, 5, 6]]
)
print(x_2, type(x_2))
```

```
[[1 2 3]
 [4 5 6]] <class 'numpy.ndarray'>
```

[5]:
```
print('x_2.shape', x_2.shape)  # 2 stands for the number of rows, 3 for the␣
 ↪number of columns
print('x_2.size', x_2.size)  # 6 represents the total number of elements
print('x_2.ndim', x_2.ndim)  # 2 represents the number of dimensions
```

```
x_2.shape (2, 3)
x_2.size 6
x_2.ndim 2
```

NumPy has a lot of functionality which we will not cover in this course. However, below we provide a few examples that are representative enough:

[6]:
```
zero_array = np.zeros((3, 4))  # create an array of size 3x4 filled with zeros
print(zero_array)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

[7]:
```
# calculate the exponential value of each element in the array
# with the base e (mathematical constant)
exp_x_2 = np.exp(x_2)
print(exp_x_2)
```

```
[[  2.71828183   7.3890561   20.08553692]
 [ 54.59815003 148.4131591  403.42879349]]
```

[8]:
```
# calculate the sum of elements in the array across columns (axis 1)
sum_axis_0 = np.sum(x_2, axis=0)
print('Sum across rows = ', sum_axis_0)

sum_axis_1 = np.sum(x_2, axis=1)
print('Sum across columns = ', sum_axis_1)
```

```
Sum across rows =  [5 7 9]
Sum across columns =  [ 6 15]
```

NumPy is useful for various linear algebra tasks, e.g., matrix multiplication, eigenvalue decomposition, etc. To know more about NumPy functionality, please see this link.

[9]:
```
print('x_1 * x_1 =', x_1 * x_1)  # element-wise multiplication
print('x_2@x_1 = ', x_2@x_1)  # matrix multiplication, equivalent to .matmull()
```

```
x_1 * x_1 = [1 4 9]
x_2@x_1 =  [14 32]
```

1.2. Pandas dataframes Pandas dataframe is a two-dimensional cunstructor for tabular data. It stores the data, allows performing various operations with it, including cleaning the data and processing it.

[10]:
```
# Create a dataframe from np.array
df_from_array = pd.DataFrame(x_2)
df_from_array.head()  # visualize the first n rows (default n=5)
```

[10]:

	0	1	2
0	1	2	3
1	4	5	6

[11]:
```
df_from_array.columns = ['one', 'two', 'three']  # set column names
df_from_array.head()
```

[11]:

	one	two	three
0	1	2	3

```
1    4    5      6
```

[12]:
```
# Create a dataframe from dictionary
df_from_array = pd.DataFrame(
    {
        'one': [1, 4],
        'two': [2, 5],
        'three': [3, 6]
    }
)
df_from_array.head()
```

[12]:
```
   one  two  three
0    1    2      3
1    4    5      6
```

[13]:
```
df_from_array['four'] = [4, 7]  # add a new column

# create a column where new value is equal to
# the sum of the value in column one and column three:
df_from_array['sum_one_and_three'] = df_from_array.loc[:, 'one'] +␣
 ↪df_from_array.loc[:, 'three']

# create a column where new value is log10 transformed
df_from_array['log10_one'] = np.log10(df_from_array.loc[:, 'one'])

df_from_array.head()
```

[13]:
```
   one  two  three  four  sum_one_and_three  log10_one
0    1    2      3     4                  4    0.00000
1    4    5      6     7                 10    0.60206
```

To know more about Pandas functionality, check out this link.

0.0.3 2. Reading files

[14]:
```
# define the core path
core_path = '/data/gardeux/Course_GG'
```

2.1. Optional: Reading files line by line and writing to files You can read files in Python with the complex of `open()`, `readline()`, `close()` functions in the following way:

[15]:
```
f_in = open(
    os.path.join(core_path, 'iris_dataset.csv'),  # path to file
```

```
    mode='r'  # mode for opening the file (see function details for more␣
 ↪options)
)
```

[16]:
```
print(f_in.readline())  # outputs the first line of the file
print(f_in.readline())  # outputs the second line of the file
print(f_in.readline())  # outputs the second line of the file
```

```
sepal_length,sepal_width,petal_length,petal_width,variety

5.1,3.5,1.4,.2,Setosa

4.9,3,1.4,.2,Setosa
```

[17]:
```
f_in.close()  # it's necessary to close the file after you've finished␣
 ↪processing it
```

Alternatively, you can use the function `.readlines()` that would provide the list of all lines contained in the file, where each line is represented as string with `\n` in the end.

[18]:
```
f_in = open(
    os.path.join(core_path, 'iris_dataset.csv'),  # path to file
    mode='r'  # mode for opening the file (see function details for more␣
 ↪options)
)
i = 0
for line in list(f_in.readlines()):  # iterate over each line
    # replace \n with an empty string and
    # split the line based on the separator
    print(line.replace('\n', '').split(','))
    i += 1
    if i == 3:  # to print only the first three lines
        break
f_in.close()  # close the file
```

```
['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'variety']
['5.1', '3.5', '1.4', '.2', 'Setosa']
['4.9', '3', '1.4', '.2', 'Setosa']
```

Now, let's try writing only the data for Setosa species into the new file named `setosa_dataset_write.csv`:

[19]:
```
f_in = open(
    os.path.join(core_path, 'iris_dataset.csv'),  # path to file
    mode='r'  # mode for reading the file (see function details for more␣
 ↪options)
)
```

```
f_out = open(
    os.path.join(core_path, 'setosa_dataset_write.csv'),  # path to file
    mode='w'  # mode for writing the file (see function details for more␣
 ↪options)
)
for i, line in enumerate(list(f_in.readlines())):  # iterate over each line in␣
 ↪the input file will all data
    # replace \n with an empty string and
    # split the line based on the separator
    values_in_line = line.replace('\n', '').split(',')
    if i == 0:
        f_out.write(line)  # write the header to the output file
    if values_in_line[-1] == 'Setosa':  # if the last element in the list is␣
 ↪Setosa
        f_out.write(line)  # write this line to the output file
f_in.close()  # close the input file
f_out.close()  # close the output file
```

2.2. Reading and writing to files with Pandas One of the most standard ways of reading tabular data is to use the `read_csv()` function from Pandas.

[20]:
```
# We use the pd.read_csv() function from Pandas to read files
iris_df = pd.read_csv(
    os.path.join(core_path, 'iris_dataset.csv'),  # specify path to file
    sep=','  # specify separator (default is comma)
)
```

[21]:
```
iris_df.head()
```

[21]:

	sepal_length	sepal_width	petal_length	petal_width	variety
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa

Let's say we want to extract only `Setosa` species and save the respective dataframe to the new file 'setosa_dataset.csv'. This can be achieved with the `.to_csv()` function in the following way:

[22]:
```
# Subset only setosa species from the dataframe
setosa_df = iris_df[iris_df['variety'] == 'Setosa']
```

[23]:
```
# We use the my_df.to_csv() function from Pandas to save dataframe to .csv
setosa_df.to_csv(
    os.path.join(core_path, 'setosa_dataset_pandas.csv'),  # specify path to␣
 ↪file
```

```
    sep=',',  # specify separator (default is comma)
    index=False,  # indication not to write index
    header=True  # indication to write header (column names)
)
```

As you may notice, reading and writing files with Pandas is easier as compared to the `open/write/close` version described in the subsection 1.1. However, when dealing with large files or files with non dataframe-like content, the latter option might be more convenient.

Side note: *NumPy arrays, dictionaries and pickled objects can be efficiently stored with NumPy to .npy, .npz and .pickle formats with the* ***`np.save()`*** *function, and read afterwards with the* ***`np.load()`*** *function.*

0.0.4 3. Analyzing and visualizing the data

In this section, we will do some exploratory data analysis (aka EDA) and visualization for the iris dataset.

[24]:
```
iris_df.head()
```

[24]:

	sepal_length	sepal_width	petal_length	petal_width	variety
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa

First, let's start with summary statistics for features of the dataset:

[25]:
```
iris_df.describe()
```

[25]:

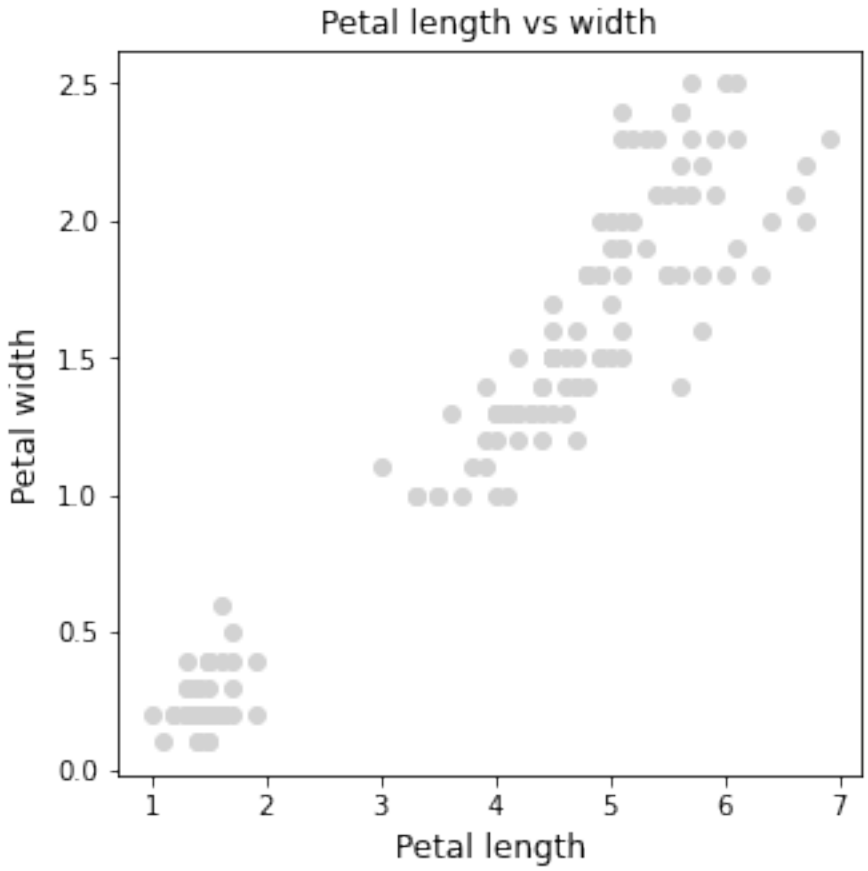
	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Exercise 1. Look into the output of the `.describe()` function. What feature has the largest mean value? What about standard deviation?

[26]:
```
# write your answer here
```

Let's check if there is any dependency between the lengths and widths of petals and sepals by using the `.scatter()` plot function from the `matplotlib` library. First, we create the figure object by definging figure (`fig`) and axes (`ax`). Axes can be either a single object, or an array of Axes objects if several subplots are created. Check out the documentation to know more about creating figures with matplotlib.

[27]:
```
fig, ax = plt.subplots(1, 1, figsize=(5, 5))  # initialize a figure
ax.scatter(
    iris_df['petal_length'],  # plot petal length on the x axis
    iris_df['petal_width'],  # plot petal width on the y axis
    color='lightgray'
)
ax.set_xlabel('Petal length', size=12)  # annotate x axis, specify the text size
ax.set_ylabel('Petal width', size=12)  # annotate y axis
plt.title('Petal length vs width', size=12)  # add a title to the plot and␣
↪specify its size
plt.show()
```



There is a clear linear dependency bertween the two features. Let's get some numerical value for this dependency by calculating Pearson correlation with the `.pearsonr()` function from the

`scipy.stats` package:

[28]:
```
correlation, corr_pvalue = scipy.stats.pearsonr(
    iris_df['petal_length'],
    iris_df['petal_width']
)
if corr_pvalue < 0.05:
    print(
        'Correlation =', correlation,
        'with the respective pvalue =', corr_pvalue)
```

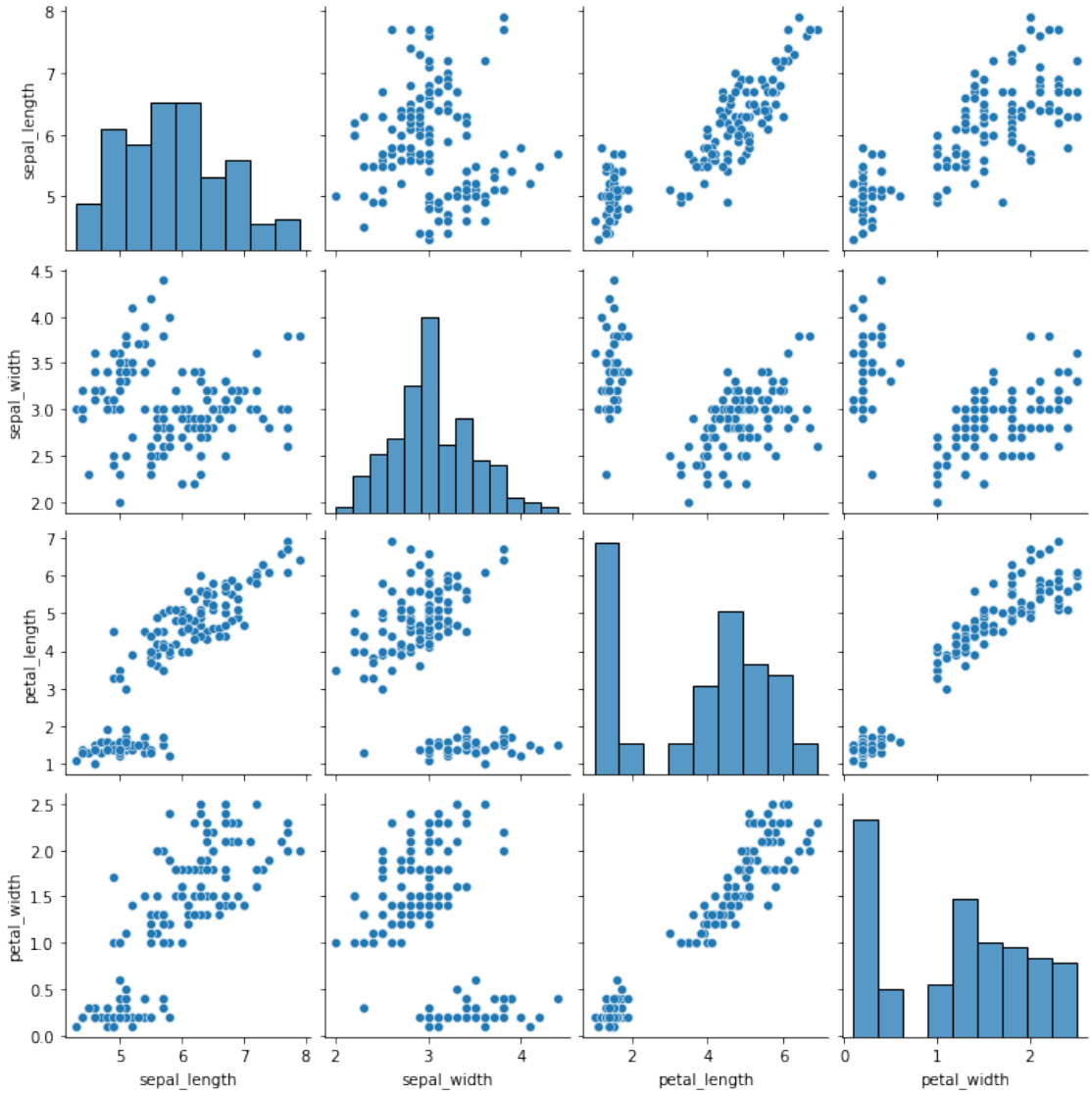
```
Correlation = 0.9628654314027961 with the respective pvalue =
4.6750039073275495e-86
```

Side note: *do not forget to check function parameters before using it!*

Since the number of features and samples is not huge, we can look into all possible pairwise relationships between the features in the dataset with the `.pairplot()` function from the `seaborn` library:

[29]:
```
sns.pairplot(iris_df)
```

[29]:
```
<seaborn.axisgrid.PairGrid at 0x1537645ada30>
```



Exercise 2. 1. How do you call the plots visualized on the diadonal of the output figure? What do they represent? 2. Look into parameters of the `.pairplot()` function and find a way how to color the data based on the iris species (the last column of the `iris_df`).

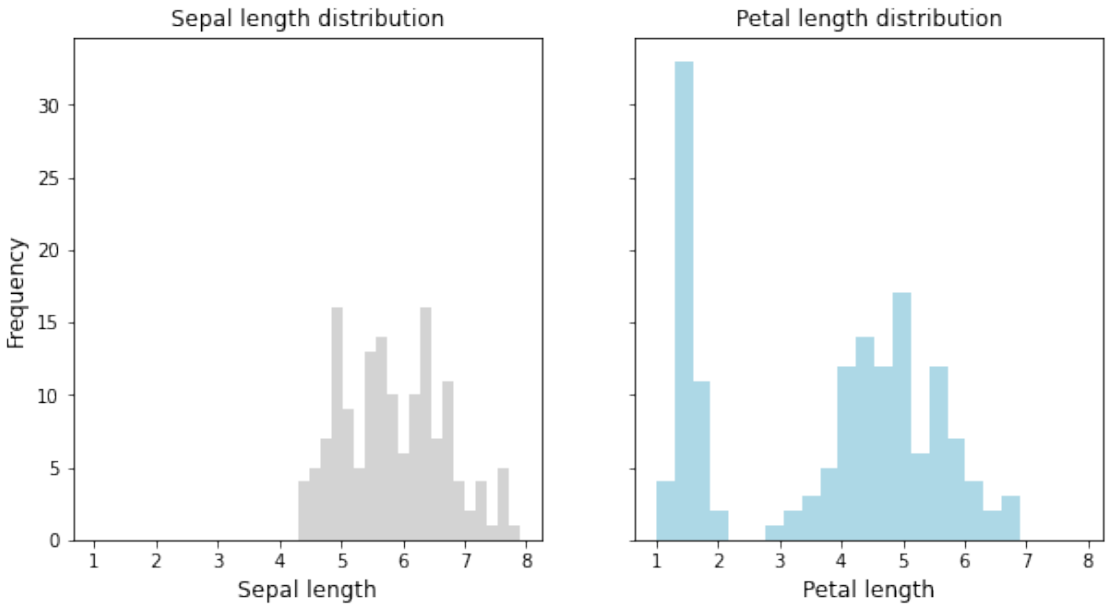
[30]:
```
# provide answers and the code in this cell
```

Let's look into the sepal and petal length distributions and compare them side by side. To visualize figures side by side, we will use the `plt.subplots()` function with parameters `nrows=1` and `ncols=2`, indicating the number of rows and columns for the subplot grid respectively.

Be careful with the Axes! Note that since we are creating more that one plot, ax will be an array of Axes object (e.g., `ax1, ax2 = ax` in our case, or alternatively, individual axes can be extracted with indexing: `ax[0], ax[1]`, as in the example below).

[31]:
```
# Initialize a figure with two subplots sidy by side:
# fig, (ax1, ax2) = plt.subplots(1, 2)   # or as follows:
fig, ax = plt.subplots(
    1,  # number of rows with plots
    2,  # number of columns with plots
    figsize=(10, 5),  # full figure size, in our case each plot will occupy␣
 ↪half of x axis
    sharex=True,  # set x axis to be the same for two plots
    sharey=True,  # set y axis to be the same for two plots
)
# in the first subplot, visualise the distribution of sepal lengths
ax[0].hist(
    iris_df['sepal_length'],
    bins=20,
    color='lightgray',
)
ax[0].set_xlabel('Sepal length', size=12)  # annotate x axis, specify the text␣
 ↪size
ax[0].set_ylabel('Frequency', size=12)  # annotate y axis, specify the text size
ax[0].set_title('Sepal length distribution', size=12)  # add a title to the␣
 ↪plot and specify its size

# in the second subplot, visualise the distribution of petal lengths
ax[1].hist(
    iris_df['petal_length'],
    bins=20,
    color='lightblue',
)
ax[1].set_xlabel('Petal length', size=12)
ax[1].set_title('Petal length distribution', size=12)
plt.show()
```



Side note: *The list of named colors available in matplotlib can be found via this link. Custom colors can be defined in RGB, RGBA formats or with hex identifiers, see this link for more details.*

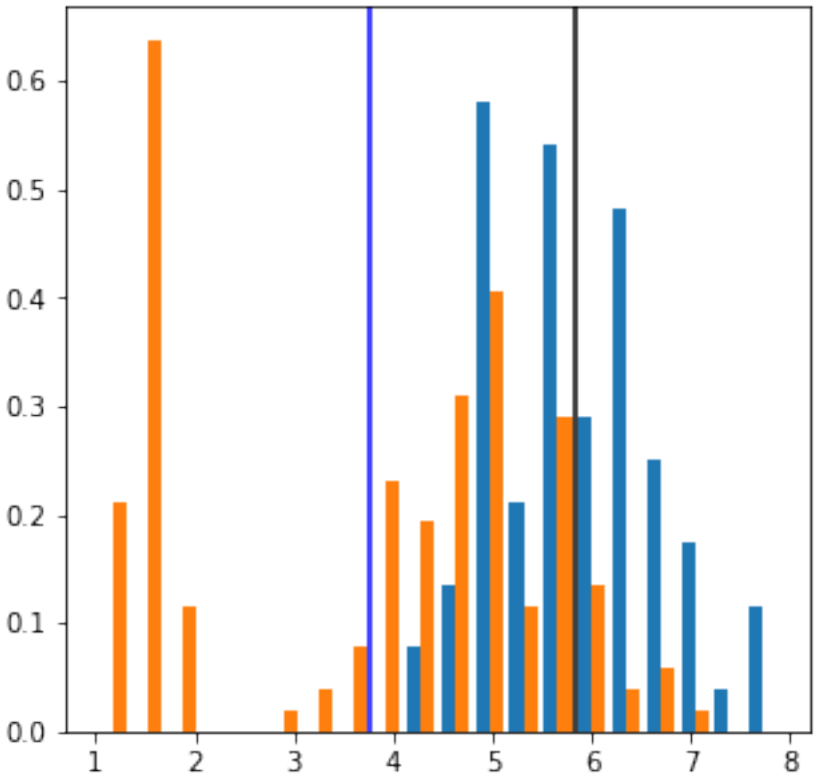
Exercise 3. let's plot the two density plots (with `.hist()` function and `density=True` parameter) in *one figure*. Complete the code in the cell below by coloring the histogram with the same colors as in the example above, set labels, add figure legend outside the axes, specify labels of x and y axes and the figure title.

[32]:
```
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.hist(
    [iris_df['sepal_length'], iris_df['petal_length']],  # two datasets in one␣
 ↪list
    bins=20,
    density=True,  # plot density value instead of frequencies
    # specify colors for the respective datasets
    # set dataset labels
)

# Indicate with vertical lines mean Sepal and Petal lengths respectively:
ax.axvline(
    np.mean(iris_df['sepal_length']),  # calculate the mean with NumPy
    color='black',
    label='Mean sepal length'
)
ax.axvline(
    np.mean(iris_df['petal_length']),  # calculate the mean with NumPy
    color='blue',
```

```
    label='Mean petal length'
)
# add x label
# add y label
# add title
# add legend

plt.show()
```



Exercise 4. We have already tried calculating correlation and the reapective p-value with the `scipy.stats` package. Now we want to know whether there is a statistically significant difference in means of sepal lengths for Setosa and Virginica species. To do that, we will use the t-test for independent samples, which is already implemented in the `scipy.stats` package. Find the respective function, use it on the sepal lengths for Setosa and Virginica species, print the value of the respective statistics and p-value. What is your conclusion?

[33]:
```
# provide the code in this cell
```

Finally, we will see how to group data with Pandas `.groupby()` function. To learn more about its

functionality, see this link.

For example, imagine we want to know the average values for all features per iris species. To do that, we will provide the column name for aggregation by specifying it as the first argument of the `.groupby()` function. This will yield the `pandas.core.groupby.generic.DataFrameGroupBy` object containing iris variety and the respecive dataframe. Later, we can use the `.mean()` function to aggregate across all samples for each feature and output the result into one dataframe:

[34]:
```
iris_df.groupby('variety').mean()
```

[34]:

variety	sepal_length	sepal_width	petal_length	petal_width
Setosa	5.006	3.428	1.462	0.246
Versicolor	5.936	2.770	4.260	1.326
Virginica	6.588	2.974	5.552	2.026

To have a better understanding of the `.groupby()` function content, let's iterate over it:

[35]:
```
for iris_variety, variety_df in iris_df.groupby('variety'):
    print('-' * 20)
    print('Iris variety is', iris_variety)
    print(variety_df.head())
    # You can process the subsetted data in any way you want
    # in this loop
```

```
--------------------
Iris variety is Setosa
   sepal_length  sepal_width  petal_length  petal_width variety
0           5.1          3.5           1.4          0.2  Setosa
1           4.9          3.0           1.4          0.2  Setosa
2           4.7          3.2           1.3          0.2  Setosa
3           4.6          3.1           1.5          0.2  Setosa
4           5.0          3.6           1.4          0.2  Setosa
--------------------
Iris variety is Versicolor
    sepal_length  sepal_width  petal_length  petal_width     variety
50           7.0          3.2           4.7          1.4  Versicolor
51           6.4          3.2           4.5          1.5  Versicolor
52           6.9          3.1           4.9          1.5  Versicolor
53           5.5          2.3           4.0          1.3  Versicolor
54           6.5          2.8           4.6          1.5  Versicolor
--------------------
Iris variety is Virginica
     sepal_length  sepal_width  petal_length  petal_width    variety
100           6.3          3.3           6.0          2.5  Virginica
101           5.8          2.7           5.1          1.9  Virginica
102           7.1          3.0           5.9          2.1  Virginica
103           6.3          2.9           5.6          1.8  Virginica
104           6.5          3.0           5.8          2.2  Virginica
```

Now you are ready for the genetics and genomics exercises!