Faculty of Mechanical Engineering and Robotics

Department of Robotics and Mechatronics

Python for Machine Learning and Data Science

Course for Mechatronic Engineering

Instruction 1:

Introduction to Python
and
Transformation and visualization of datasets

You will learn: How to work with Google Colab and the basics of Python. How to load, transform, visualize data and finally how to split data into train, test, and validation sets. These will be used extensively throughout this course.

Additional materials:

- Course lecture 1 [*obligatory*] http://galaxy.agh.edu.pl/~zdw/Materials/Python/LectureNotes/
- Matlab to Python handout http://galaxy.agh.edu.pl/~zdw/Materials/Python/Matlab%20-%20python%20handout.pdf

Learning outcomes supported by this instruction:
IMA1A_W07, IMA1A_W12, IMA1A_U05, IMA1A_U07, IMA1A_U14

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Adam Machynia, Mateusz Heesch, machynia@agh.edu.pl

Initial information regarding course laboratories

Laboratory structure

The laboratories are prepared using a *Reversed Classroom* method. It means that the intended way of doing them is to first do part of them at home and then finalize work during classes. The more you do at home, the more knowledge you'll gain and (hopefully) the easier the entire course will be for you.

The general flow of each instruction includes an explanation of the steps that should be taken to solve a problem [in a white background], followed by a series of tasks for you to complete. These tasks are divided into three categories: "For 3.0 mark" (marked in red), "For 4.0" (marked in orange), and "For 5.0" (marked in green). Completing a set of tasks during the laboratory results in a conditional mark. For the next laboratory, you should finish all exercises for 3.0 and 4.0 marks, which are always mandatory. After you show and defend the complete set of exercises for 3.0 and 4.0 marks during the next classes, you will receive a final mark. Tasks for 5.0 are not obligatory. In other words, the more tasks you complete during classes, the higher your grade will be – but you will need to complete most of them either way.

Correcting absence and laboratory fails

The intended way of getting a laboratory pass can be disturbed in three ways:

- You can be absent (then you won't have a chance to get a 'conditional grade')
- You can fail to do tasks "for 3.0" during the laboratory
- You can fail to defend the conditional grade during the next laboratory

In all of these cases you will be required to prepare a laboratory report including all the necessary tasks (required for 3.0 and 4.0 to get 4.0 mark and also 5.0 tasks to get 5.0) plus an additional one selected by the teacher from the additional tasks pool available at the end of each laboratory. You will be required to defend this report.

Note, that you can pass the laboratory in such way only two times during the entire course. If you have more laboratories to correct, you will need consultations with the course coordinator who will individually assess the situation and will either determine the scope of work required to get a pass or direct you for taking the course again.

Tests and test corrections

Some laboratories will start with a lecture test. The schedule for tests is available in notes from the first lecture. For each test there will be two correction chances at the end of the course. If, however, you fail more than two tests during the course, the correction chance will involve entire lecture material.

Introduction

Throughout the course you will be using the google Colab environment for writing and running your python codes – this decision was grounded mostly in ease of set-up and use, as well as holding some similarity with operating in Matlab, which you are all familiar with.

Further, you will get familiar with some Python libraries: Pandas, Matplotlib, and Seaborn. Data visualization using different types of plots will be introduced. You will work with bar graphs, histograms, scatterplots, and boxplots.

Getting started with google Colab

First of all – what is google Colab? It's a cloud-based jupyter notebook environment that requires no set-up, making it a perfect candidate for starting work with python.

To start working with Colab go to the link below (note – requires a google account, it also has payment plans for higher performance/functionality packages, none of which will be required by this course)

https://colab.research.google.com

If you do not have a google account, create one here:

https://accounts.google.com/signup

After getting to the landing page, you can familiarize yourself with the introduction and basic functionality tutorials on the page. To open your own "notebook" go to top left corner and click File -> New notebook.

If you want to switch the display language from whichever language your google account is set to, to english, go to "Help -> Display in english"

Your first notebook will start empty, with a single code cell present on top, like in Figure 1. New code cells will be added automatically as you run cells, or can alternatively be added ahead of time by hovering mouse around the center point of the cell, on either lower or upper edge.

Running these cells (either by pressing the run icon on the left side of the cell, or shift+enter shortcut while in the cell) will execute the code inside, and at the end of the execution also print the output of the last line if it returns a value(see Figure 2). The values assigned to variables are kept in memory and are preserved between cells (see last cell in Figure 2).

Task 1.1: Perform 2 simple arithmetic operations in Colab – display the result of one of them via print() function, and the other via in-built display – See Figure 2

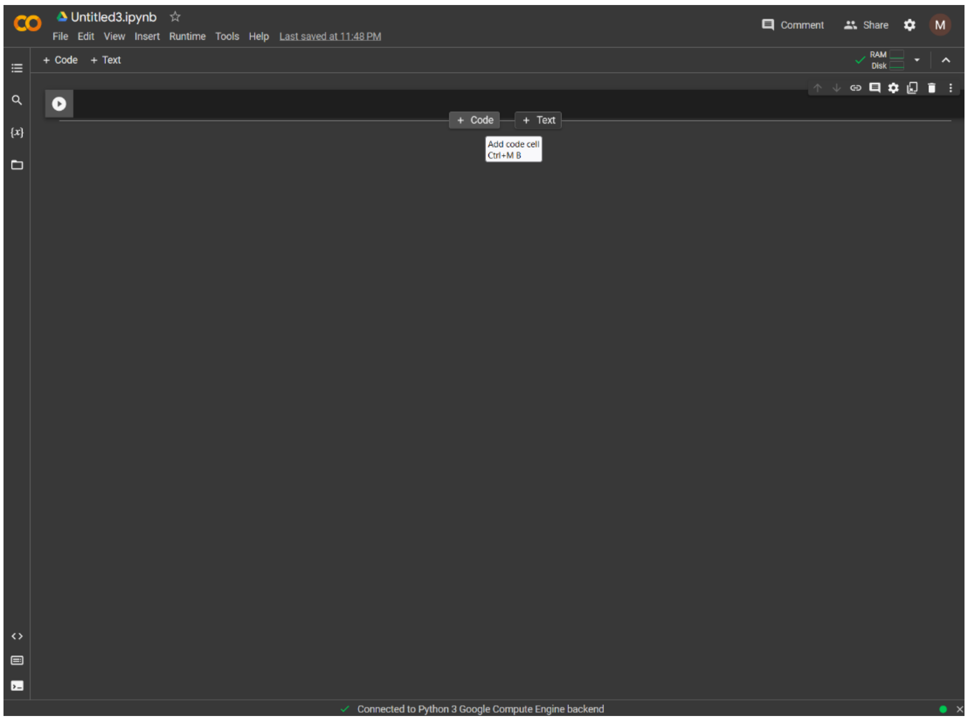
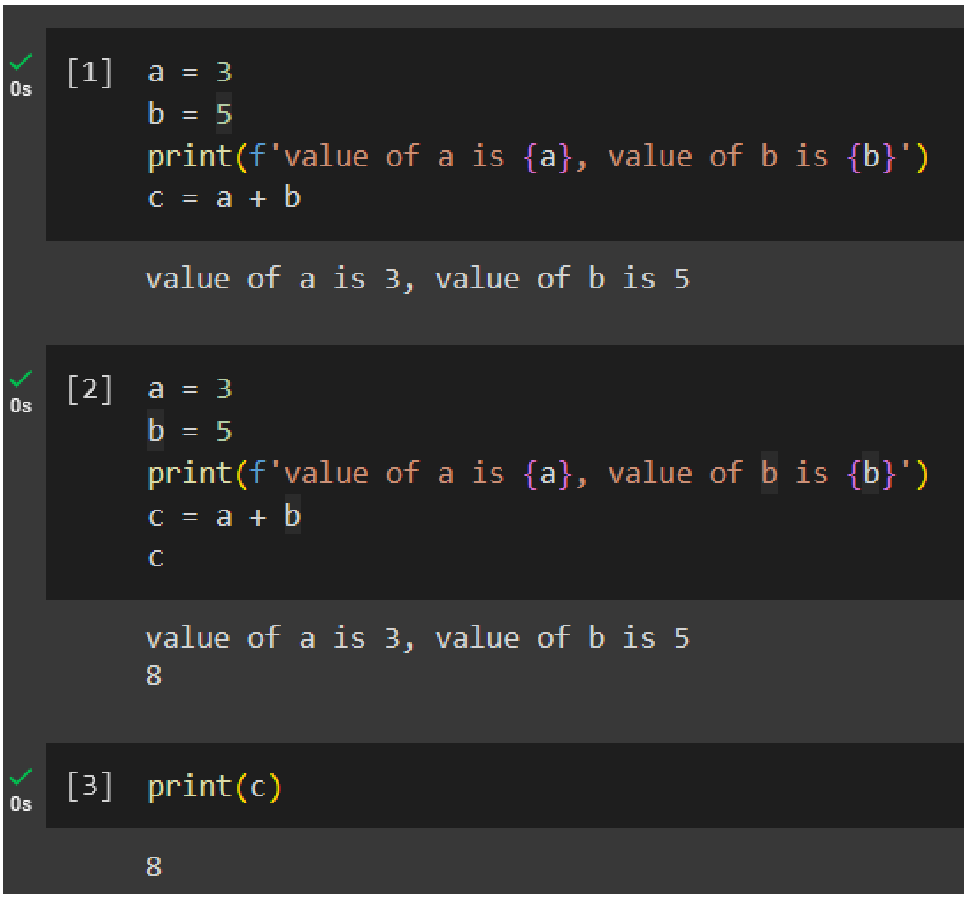

Untitled3.ipynb
File Edit View Insert Runtime Tools Help Last saved at 11:48 PM
Comment
Share
M
+ Code + Text
RAM
Disk
+ Code
+ Text
Add code cell
Ctrl+M B
Connected to Python 3 Google Compute Engine backend

Figure 1 – Empty notebook



```
[1] a = 3
    b = 5
    print(f'value of a is {a}, value of b is {b}')
    c = a + b
```

```
value of a is 3, value of b is 5
```

```
[2] a = 3
    b = 5
    print(f'value of a is {a}, value of b is {b}')
    c = a + b
    c
```

```
value of a is 3, value of b is 5
8
```

```
[3] print(c)
```

```
8
```

Figure 2 – example of cell outputs

Basic python syntax

Feel encouraged to put these codes, as well as codes from your matlab->python handout into your Colab notebook and run them, especially if you're not sure how exactly they work and want to examine their behavior.

As you could see in the example in Figure 2, defining numerical variables and performing arithmetical operations on them functions identically to matlab, printing in this example was done with an f-string, which allows us to inject values from variables mid-string conveniently. It's syntax is straightforward and looks as follows:

```
f'some text {variable_name}, some more text'
```

Where {variable_name} evaluates the value in that variable and converts it to string.

Besides playing around with variables and printing their values, we may also want to use conditional statements, like in following example:

```
import random #importing library for random number generation
num = random.random() #creating a random number, should be in <0, 1> range
num2 = random.random()

if (num < 0) or (num > 1): # example conditional with "or"
    print('that shouldnt happen')
elif num == 0.5: # example equality conditional on "else if" block
    print('exact half, lucky')
elif (num > 0.5) and (num2 > 0.5): # example conditional with "and"
    print('yes, num2 was defined only to have a case for "and"')
else:
print('no lucky half, and either num or num1 (or both) are lower than 0.5')
```

The above code demonstrates a simple if-elseif-else switch based on the value of a randomly generated number.

We'll also be using *while* loops:

```
import random #importing library for random number generation

while random.random() < 0.9: # will be evaluated at start of every loop
    print('rolling for value greater or equal to 0.9 failed, rerolling')
    print('this will also be executed in each loop iteration')
print('finally out of the loop!') # will happen after loop is done
```

Note that no brackets or keywords are used to control what's inside the loop. Instead, code structure/hierarchy in python is controlled via indentations (indentation level being a quadruple space, usually done by pressing tab). In this case, the first 2 prints are on the same indentation level, corresponding to that inside the defined *while* loop.

And lastly, *for loops:*

```
# usual syntax
for i in range(10):
    print(i)

# iterating over list
days = ['monday', 'tuesday', 'wednesday', 'thursday',
        'friday', 'saturday', 'sunday']
for day in days:
    print(day)
```

In general, similarly to matlab, the *for* loop iterates over a collection of data, assigning new piece of this data to a specified variable at each loop iteration. In the "usual syntax" example, the range() function is used, which creates a collection of integers in ascending order, starting at 0 and ending at specified value (non-inclusive, e.g. range(3) will have [0, 1, 2]).

Task 1. 2: Similarly to the "while" loop in section 3, write a "while" loop based on the value of randomly generated number being higher than 0.8, add following functionality:
1) After the loop ends, print how many iterations it went through
2) Based on naive expected number of iterations (e.g. for 0.5 we "expect" 1 pass for every 2 numbers, so needing more than 2 would be "unlucky"), after the loop ends print "lucky" or "unlucky" message.

Functions and classes

Function syntax starts with "def", followed by function name, function arguments in brackets, and then lastly a colon. The function *usually* ends with the "return" line, which decides what value is returned when this function is called in code, however not all functions *need* to return something.

```
def sum_values(a, b=0):
    c = a + b
    return c
```

You can assign a default value to an argument, allowing further uses of the function to omit it if changing it is not necessary, such as the "b" in example above.

Task 1.3: Put the loop from task 2 inside a function, where the threshold for random number will be an input argument to the function. Run the function with different argument values.

Task 1.4: Write a program that solves the following equation (the solution can be approximate):
3x2 – 4x - 2 = 0.

Task 1.5: Write a function for solving quadratic equations, use it to solve task 1.4.

Creating a class

```
from math import sqrt
class TwoDCar:
    def __init__(self, color='red'):
        self.position = [0, 0]
        self.distance_traveled = 0
        self.color = color

    def move2D(self, move_vector):
        self.position[0] = self.position[0] + move_vector[0]
        self.position[1] = self.position[1] + move_vector[1]
        dist = sqrt(move_vector[0]**2 + move_vector[1]**2)
        self.distance_traveled = self.distance_traveled + dist
```

Classes are "template" definitions which contain some data (parameters) and are able to do some things (methods). Class should always have the "__init__(self)" function which is the constructor – the function called when the class is instantiated (see next section). Besides the constructor, a class can have any number of methods depending on the needs. In general all methods should have "self" as the first argument (not entirely true, but for simplicity's sake let's stick to that), which denotes the object itself, and is used to access its parameters and methods.

creating a class instance (object)

```
car_default = TwoDCar()
car_blue = TwoDCar(color='blue') # overwrites default value of 'color' with 'blue'
```

interacting with an object

```
car_blue.move2D([10, 0]) # will call the move2D method of the object
car_blue.move2D([5, 5])

print(car_blue.position) # will access the position parameter, with expected value of
[15, 5]
print(car_blue.distance_traveled) # will access distance_traveled parameter, with
expected value of ~17.07
car_blue.distance_traveled = 0 # will reset the value of distance_traveled parameter
```

Longer code examples

Use this for reference on how various things work and might be used – feel encouraged to paste this code into Colab and examine what it does.

```
import numpy as np
import matplotlib.pyplot as plt

def rectangle_circumference(a, b):
    return 2*a + 2*b

def rectangle_area(a, b):
    return a*b

value_count = 20

# initializing vectors of 20 random values scaled from 0 to 1
random_sides_a = np.random.rand(value_count)
random_sides_b = np.random.rand(value_count)

# scaling and shifting the vectors to range from 1 to 10
random_sides_a = (random_sides_a * 9) + 1
random_sides_b = (random_sides_b * 9) + 1

# initializing empty lists
circumferences = []
areas = []

for i in range(value_count):
    # appending values to lists
    circumferences.append(rectangle_circumference(random_sides_a[i],
                                                  random_sides_b[i]))
    areas.append(rectangle_area(random_sides_a[i], random_sides_b[i]))

# utilizing zip to iterate over several collections at the same time
for a, b, circ, area in zip(random_sides_a, random_sides_b,
circumferences, areas):
    print(f'sides {a} and {b}, circumference {circ}, area {area}')

# printing newline to separate output logs
print('\n')
# obtaining order of indices by ascending value
order = np.argsort(areas)

# utilizing enumerate() to get a helper variable for iteration number
for iteration_no, index in enumerate(order):
    print(f'{iteration_no} place, rectangle #{index}, area {areas[index]}')

plt.plot(sorted(areas))
plt.title('distribution of areas of rectangles')
plt.xlabel('sample #')
plt.ylabel('area [units^2]')
plt.grid()
plt.show()

plt.plot(sorted(circumferences))
plt.title('distribution of circumferences of rectangles')
plt.xlabel('sample #')
plt.ylabel('circumference [units^2]')
plt.grid()
plt.show()
```

Task 1. 6: Given following "input" code (paste it into first cell and execute it):

```
import numpy as np
values1 = np.random.rand(20)
values2 = np.random.rand(20)
```

Write code that separates the values in both values1 and values2 arrays into two lists (for information on how to use lists, refer to Example code in section *Longer code examples*, both "circumferences" and "areas" are lists), one containing values lower than 0.5, the other containing remaining values, and:

1) Display the contents of these lists (you can print an entire list - print(list_name))

2) Display the number of elements in these lists (you can use len() function).

Dataset loading

We will use the following dataset from UCI Machine Learning Repository.

https://archive.ics.uci.edu/dataset/186/wine+quality

In Colab there is a simple "Files" window (Figure 3) where you can manage files. Simple drag and drop will work, however, in this case, the file will be stored only for a current session. So, next time you will need to reload it. Another possibility is to connect Colab with Google Drive.

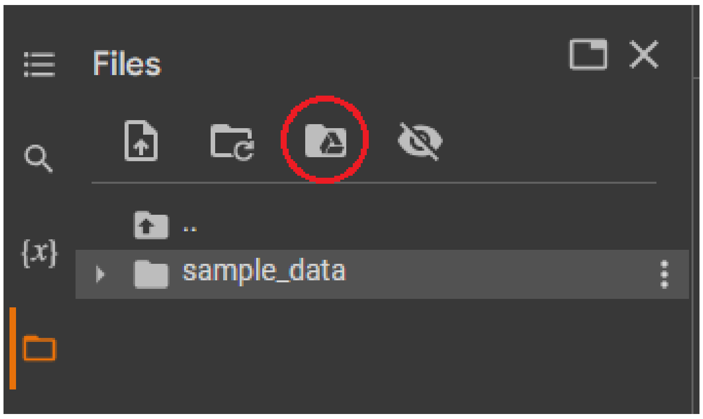


Figure 3 – Files in Colab, "Mount Drive" button marked.

Task 1.7: Go to the UCI MLR website and download the CSV files containing the dataset. Then, place the files in your Google Drive and connect Colab with it.

Simple way to read CSV files into Python is using Pandas library and its function *read_csv*. Note that if in the CSV file the semicolon is used instead of a comma, you will have to specify the proper data separator while loading the dataset.

```
import pandas as pd
data = pd.read_csv('path/file.csv')
data = pd.read_csv('path/file.csv', sep = ';') # When data separator needed
```

This will result in creating a data structure called DataFrame. It is a two-dimensional data structure similar to spreadsheets. It consists of columns and rows, see Figure 2. Generally, columns match features in the dataset and rows are particular observations. What is important, DataFrame can store not only numerical data.

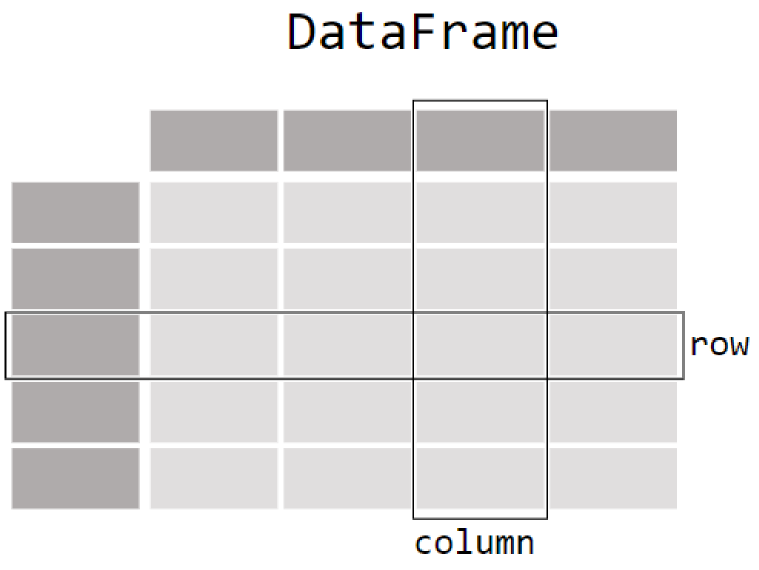


Figure 4 – Pandas *DataFrame* (source).

The following methods give a brief description of the dataset.

`data.head()` – prints five first rows of a DataFrame.

`data.describe()` – prints some basic statistics describing numerical data: number of observations, mean, standard deviation, minimum, maximum, and percentiles.

Task 1.8: Load the dataset from a CSV file using Pandas, and examine its contents using the *head* and *describe* functions. Start with the *red* part of the dataset.

Hint: right-click on a selected file and copy the path. This will show you how to reference files stored in Google Drive.

Take a quick look here.
https://pandas.pydata.org/docs/getting_started/intro_tutorials/01_table_oriented.html
This will help you understand what's going on with your data.

Data visualization

Having loaded the dataset we want to prepare some charts to get an insight into our data. It is possible to create some graphs directly from DataFrame using method *plot()* or *plot.name()*, where name stands for specific plots' type: `'area'`, `'bar'`, `'barh'`, `'box'`, `'density'`, `'hexbin'`, `hist'`, `kde'`, `'line'`, `'pie'`, `'scatter'`. For example, to create a simple histogram we might use the following code.

```
data['quality'].plot.hist()
plt.title('Quality distribution')
plt.xlabel('Quality')
plt.ylabel('Count')
```

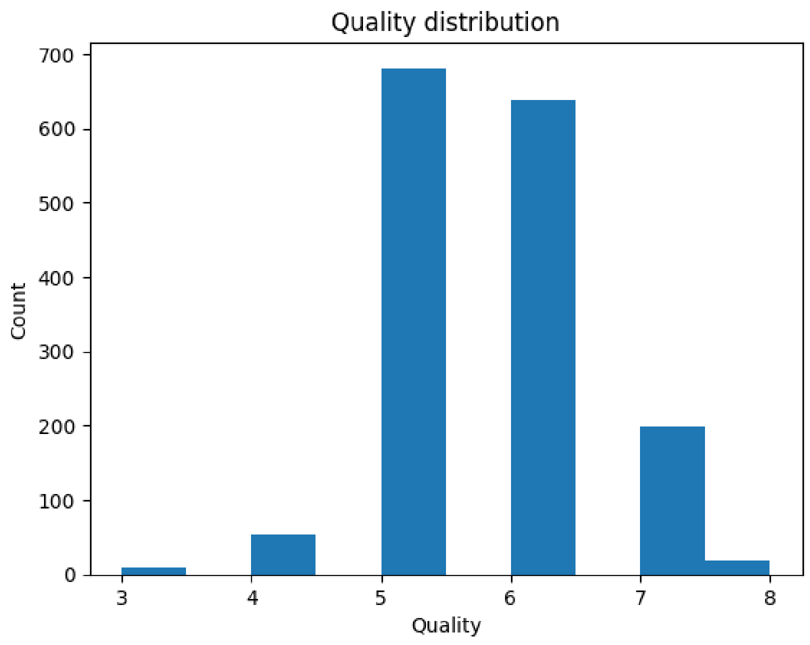


Figure 5 – Example of a histogram.

This simple plotting style sometimes might get doubtful as it works only for numerical data. What if we want to plot the histogram showing the number of observations regarding a specific variable or check if any categorical variable is well-balanced? DataFrame columns can be indexed as Python's dictionaries, thus returning so-called Series, which are one-dimensional ndarrays with labels (see more: https://pandas.pydata.org/docs/reference/api/pandas.Series.html). The following example will create a histogram that is the same as previously (Figure 5) but notice that now we use only direct matplotlib methods for plotting.

```
quality = data['quality']
plt.hist(quality)
plt.title('Quality distribution')
plt.xlabel('Quality')
plt.ylabel('Count')
```

Task 1.9: Create the following plots.
1) A histogram showing the number of instances of a specific quality.
2) Scatterplot showing the citric acid vs fixed acidity.
3) Any boxplot.
Please remember to label axes, print titles, and so on.

To save a figure, the method *savefig* can be used. Most conveniently use it on the figure object (example below: *f1*). You can specify easily the file format by adding a proper suffix to the name which is the first parameter e.g. *'name.png'.* Proper file formats are png, pdf, svg, and eps. To specify image resolution use parameter *dpi,* otherwise, it will be saved in a resolution of a plotted figure.

```
f1 = plt.figure()
plt.hist(quality)
f1.savefig('name.png', dpi=600)
```

Task 1.10: Save the selected plot as a PNG file using the *savefig* method.

Task 1.11: Not all created plots look pretty. Try using customization in matplotlib.pyplot to make them eye-catching albeit still clear.

Let's use Seaborn. It will be helpful for us mostly for two purposes: it strictly cooperates with DataFrames and makes plots look a bit nicer.

```
# Import seaborn
import seaborn as sns
# Apply the default theme
sns.set_theme()
```

Generally, using Seaborn is similar to using matplotlib. You can create a scatterplot (and any other type of plot) in two ways.

```
sns.scatterplot(x=data['pH'], y=data['density'])
sns.scatterplot(data = data, x='pH', y='density')
```

And here is a hint for simple figure size management.

```
plt.figure(figsize=(10, 5))
```

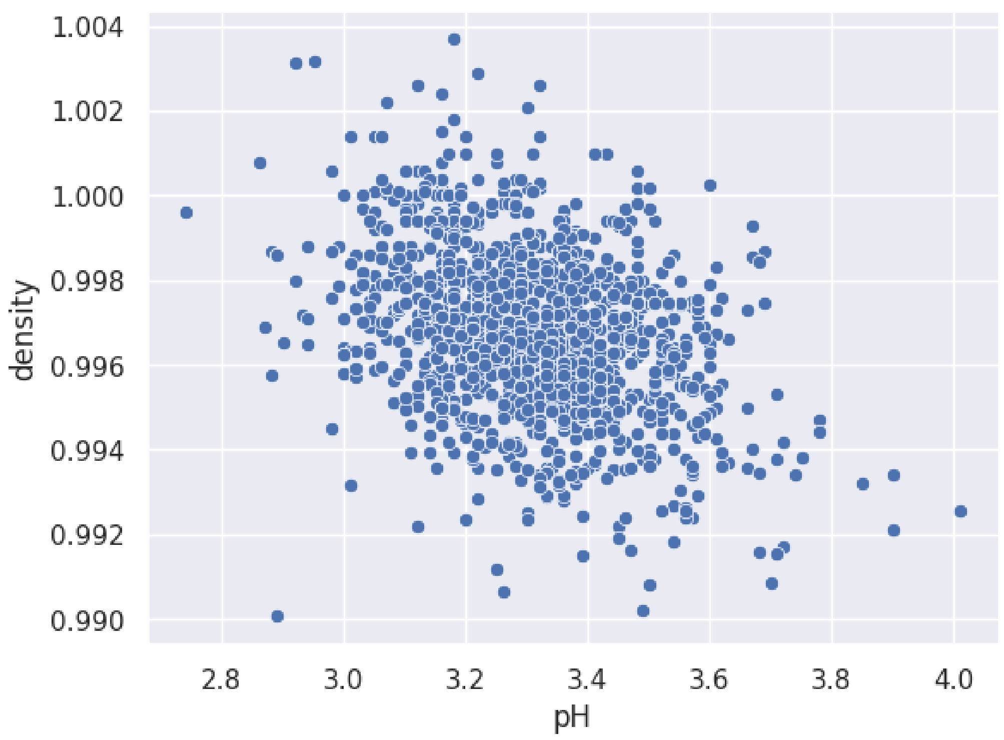


Figure 6 – Example of a scatterplot.

The simple scatterplot as in Figure 6 might be very informative, however, we can add some more details. Let's add information about the quality to this visualization, so it would look like in Figure 7.

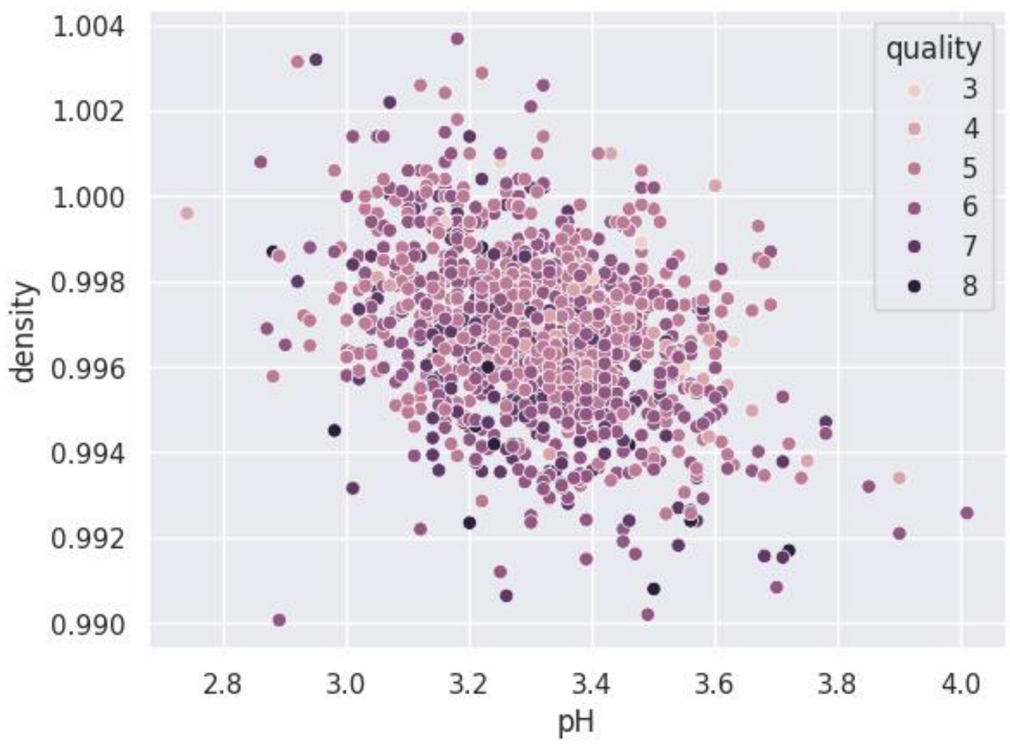


Figure 7 – Second example of a scatterplot.

Task 1.12: Having code to create a simple scatterplot (Figure 6), improve it to have a categorical scatterplot (Figure 7).

Task 1.13:

1) Check how your plots look after applying Seaborn's default theme.

2) Create a boxplot showing density for each particular quality score.

Hint: `sns.boxplot(x=?, y=?)`

Task 1.14: Correlation heatmap.

You can calculate correlation simply by:

`corr = data.corr().`

This method will automatically handle nonnumerical data. Then, to plot it use:

`sns.heatmap(corr).`

This visualization will not be perfect. Choose a proper colormap and turn on annotations.

Data split: training, validation, test

An important issue that we will have to handle before creating machine learning models is splitting data into subsets. This step aims to ensure that models and their parameters will be properly verified. Thus, we will split the data into three subsets.

1. **Training set.** The model will be trained on this part of the data.
2. **Validation set.** The model's parameters will be verified on this set.
3. **Test set.** This part is used for final verification and comparing different models.

Commonly the split is 60-80% training data, 10-20% validation data, and 10-20% test data.

To address this task we will use the *scikit-learn* library and function *train_test_split.*

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Parameters are X – features on which predictions will be based, y – target, and test_size – the ratio of test set size. So before using this method, we need to point target in our dataset. Let's assume that we want to predict the quality based on all other features.

```
X = data.drop('quality', axis = 1)
y = data['quality']
```

Task 1.15: Split the dataset into the train, validation, and test sets with a ratio of 0.6:0.2:0.2. Find out how to obtain the reproducibility of this split.

Task 1.16: Think which features would be reasonable to choose for the target. Also, consider if any features should be excluded when building the prediction model. Be ready to discuss your ideas with the teacher.

Additional tasks and challenges:

Classes

Though using objects is not something you can avoid during this course (e.g. numpy array, or pandas DataFrame), writing your own ones is not necessary – it may however lead to more compact and readable code.

Task 1.17: Classes Create a "sorting" class that would solve Task 6a and 6b. Its parameters should include the two empty sorting lists, and it should have a one method taking in a one-dimensional collection of values as input, sorting them into the two internal buckets, and another method for displaying contents and lengths of these buckets. Try instantiating (creating an object instance) and using it.

Optimizing via matrix operations.

Overall python is not known for it's speed – quite to the contrary, it's a rather slow language. However, there are ways to make it run much faster than it has any right to. One of the more relevant ones that you'll run into during this course is utilizing matrix operations. Numpy library is based around C-bindings, meaning any operation done with it should be fairly close in speed to a pure C implementation – meaning that even more so than in matlab, we want to utilize matrix operations as much as possible.

E.g, adding noise to an arbitrary signal:

We could do it as follows:

```
import random
signal_vector = get_signal() # creating signal in some arbitrary way
for i in range(signal_vector.shape[0]):
    signal_vector[i] = signal_vector[i] + random.random()
```

OR we could do it like that:

```
import numpy as np

signal_vector = np.array(get_signal()) # creating signal in some arbitrary way
noise_vector = np.random.rand(signal_vector.shape[0])
signal_vector = signal_vector + noise_vector
```

Where instead of adding values one by one in a loop, we simply add two entire vectors at once.

Task 1.18: Matrix operations. Have a look at the code example in section *Longer code examples*, try to rewrite it (up until the printing) in such a way that no loops are utilized

List / dictionary comprehensions
Python has an in-built mechanism for optimized construction of lists and dictionaries – it will be in no way necessary to use it during the course, it's simply a convenience tool for slightly faster list construction, and more compact (albeit less readable) code.

For instance, let's take grabbing a list of files with png extension from a directory:

```
import os

path_of_interest = get_path() # getting some arbitrary path
all_files = os.listdir(path_of_interest)
filtered_files = []
for file in all_files:
    if 'png' in file:
        filtered_files.append(file)
```

Alternatively we could use list comprehension:

```
import os

path_of_interest = get_path() # getting some arbitrary path
filtered_files = [file for file in os.listdir(path_of_interest) if 'png' in file]
```

Or cut it down to 1 line if we're at it anyway:

```
import os
filtered_files = [file for file in os.listdir(get_path()) if 'png' in file]
```

The first version is by far the most readable, but it's marginally slower and takes far more typing – whether you want to use this tool or not is entirely up to you, as it's mostly a convenience vs readability tradeoff.

We can also make arithmetic operations in list comprehension, as such:

```
double_sequence = [index * 2 for index in range(10)]
```

Which will iterate over the "range" collection, with values from 0 to 9, and multiply each of them by two.

Task 1.19: List comprehension. Simplify the code below to a single line (paste the original code into Colab to compare results)

```
newlist = []
for i in range(10):
    value = i
    value = value * value
    value = value - 4
    newlist.append(value)
```

Task 1.20: Create a violin plot showing density for each particular quality score.

Task 1.21: Using *seaborn* create a line plot showing wine quality against pH. Be ready to discuss what is on the plot.

Task 1.22: Prepare similar visualizations as in the previous tasks for the white wine set.