Programming for Data Science 2023 - Assignment

Overview

Goal

The goal of the midterm assignment is to test your familiarity with the concepts you have learned during the first 5 weeks of the semester.
Above all, it is an opportunity for you to challenge yourself and practice. If you are having difficulties with the assignment, reach out for support through the Discussion Forum.

The assignment is divided into 2 parts:

1. Python basics - Explore core building blocks of programming, focusing on variables, data structures, and their manipulation
2. K-Means from scratch - Build an unsupervised learning algorithm step by step

Rules

- You can only use packages from the Python Standard Library (built-in), namely https://docs.python.org/3/library/ (this means no pandas, numpy, scikit-learn, etc.)
- Make sure that you don't have errors in your code. If you don't know how to do a specific question, leave it as it was. You should do a clean run all, in the end, to make sure all cells run without errors
- You should limit your code to fill the question function and an extra block for testing. Please avoid creating many code blocks
- Make sure your notebook name is unchanged from `python_assignment`
- Make sure that you save the requested contents in a folder named accordingly to the rules: `<student_number>_<firstname>_<lastname>`
- Make sure to zip your final folder

Grading

- This assignment will be graded from 0 to 20 points
- The assignment is worth 40% (8.00 values) of your final grade
- Each question has its corresponding score associated, e.g. `Q1 [1.0p]`
- Each question will be automatically validated against a set of scenarios and graded accordingly
- If your code does not run (e.g. you click *run all* and errors are raised in one or more questions), there will be a penalty of 0.5 points per error
- Questions that raise errors by default will not be graded

- A penalty of 0.5 points will be applied per file missing in the delivery of required content
- A penalty of 1.0 points will be applied per day late to late deliveries

Delivery

- Deadline: **02 April 2023 23:59:59**
- Format: `.zip`
- Name: `<student_number>_<firstname>_<lastname>`
- You should deliver the following contents:
    - `python_assignment.ipynb` - Your jupyter notebook assignment solved
    - `python_assignment.html` - A PDF HTML export of notebook that you can get by going to `File > Download as > PDF via HTML (.html)`
- Where: Submission is done through the respective Moodle activity
- You should not change the name of the notebook `python_assignment`

```
# Example of good submission
20230101_john_smith.zip
├── 20230101_john_smith
│   └── python_assignment.ipynb
│   └── python_assignment.html

# Another example of a good submission
20230101_john_smith.zip
├── python_assignment.ipynb
├── python_assignment.html

# Example of bad submission
# In this example the zip name is wrong
# And the notebook name is wrong
# And it does not contain all elements (missing .html)
john_smith_FINAL_FINAL_2.zip
├── john_smith_FINAL_FINAL_2
│   └── my_notebook.ipynb
```

How to

Each question consists of a function template that you have to fill in to achieve the requested solution. By making it into a function, we can automatically call your function and check it against a series of scenarios to make sure it's okay.

For example:

```
# Q1: Fill the function sum_numbers(a, b) that takes two
integers 'a' and 'b' and returns the sum of both.

def sum_numbers(a, b):
    # your code here
    pass
```

By default, each function will have the command `pass` so that nothing will happen when the function runs (and no error is raised). You should replace it with your code.
Here's how you should complete the exercise

```
def sum_numbers(a, b):
    sum_of_numbers = a + b
    return sum_of_numbers
```

And then you can test your code

```
[In]: sum_numbers(3, 2)
[Out]: 5
```

Most questions have three parts:

1. The instruction
2. An example
3. Assumptions (you can take them as truth)

For example, if the assumption of the above function is `a and b are always integers and > 0` then you know that in our test scenarios, we will never test it with something such as `sum_numbers(-5, 10)` or `sum_numbers(1.2, 3.5)` .

Imports

Import any library you require here. For example, `random` is already imported for you. Remember you are only allowed to use built-in python libraries as stated before, which can be found here: https://docs.python.org/3/library/

In [5]:
```
import random
```

Part I - Python Basics [10.0p]

Q1 [0.25p]: Create the function:

```
def solve_equation(x)
```

that, given a number `x`, returns `y` according to the following equation:

$$y = 6x^2 + 3x + 2 \qquad (1)$$

For example:

```
[In]: solve_equation(2)
[Out]: 32
```

Assume that:

- `x` is a non-empty integer or float within [-10^5, 10^5]

In [6]:
```
def solve_equation(x):
    y = 6 * (x**2) + 3 * x + 2
    return y

solve_equation(2)
```

Out[6]:
```
32
```

Q2 [0.25p]: Create the function:

```
def check_odds(numbers)
```

that, given a list `numbers` of positive integer numbers, checks if the total ammount of odd observations is higher than the total amount of even observations. If the total of odds is higher, it should return the boolean value `True`, otherwise it should return the boolean value `False`.

For example:

```
[In]: check_odds([1, 2, 3, 4, 5])
[Out]: True
```

Assume that:

- the length of `numbers` is non-empty and at most 10^5
- each N in `numbers` is an integer within [0, 10^5]
- there is at least 1 odd number in `numbers`

In [7]:
```
# A função itera a lista. Se o elemento da lista % 2 = 0, vai para a var 'even
# Depois da interação, se existirem mais elementos na 'odd' dá como output True
# Se existirem menos ou iguais elementos na 'odd' dá como output False.

def check_odds(numbers):
    odd=0
    even=0
    for i in numbers:
        if i % 2 == 0:
            even += 1
        elif i % 2 == 1:
            odd += 1
    if odd > even:
        return True
    elif odd <= even:
        return False

check_odds([1, 2, 3, 4, 5, 6])
```

Out[7]:
```
False
```

Q3 [0.5p]: Create the function:

```
generate_numbers(n)
```

that, given a positive integer `n` returns a list of n randomly sampled integers (allows replacement) between -25 (inclusive) and 25 (inclusive).

For example:

```
[In]: generate_numbers(5)
[Out]: [10, 5, 0, -5, 2]
```

In the previous example, `n=5` returns 5 randomly sampled numbers.

Assume that:

- `n` is a non-empty integer and at most 10^5

In [8]:
```
# É aberta a lista 'atoalist'. Imediatamente é feita uma iteração pelos element
# É importado o módulo Random que gera, de acordo com N elementos providenciado
# -25 e 25 para a 'atoa' list. Depois, esta última lista é adicionada à 'atoali

def generate_numbers(n):
    atoalist = []
    for i in range(n):
        import random
        atoa = random.randint(-25, 25)
        atoalist.append(atoa)
    return atoalist

generate_numbers(4)
```

Out[8]:
```
[-1, -11, 21, -11]
```

Q4 [0.5p]: Create the function:

```
def update_list(numbers)
```

that, given a list of integers `numbers`, the function uses the function `map()` to return a new list that obeys to these rules:

For each value in the input list:

- if the element is even, you should add 1
- otherwise, if the element is odd, you should subtract 1.

For example:

```
[In]: update_list([0, 1, 2, 3])
[Out]: [1, 0, 3, 2]
```

Assume that:

- the length of `numbers` is non-empty and at most 10^5
- each N in `numbers` is an integer within [-10^5, 10^5]

In [9]:
```
# 2 funções: A função 'update', se o elemento do input 'numbers' for even adic
# 1 valor. No return, a function 'map' aplica a função 'update' a todos os elem
# a function 'list' transforma o output da function 'map' em lista.

def update_list(numbers):
    def update(n):
        return n + 1 if n % 2 == 0 else n - 1
    return list(map(update, numbers))

update_list([0, 1, 2, 3])
```

Out[9]:
```
[1, 0, 3, 2]
```

Q5 [0.5p]: Create the function:

```
def get_logs10(numbers)
```

that, given a list of positive integer `numbers`, returns a new list where each element is the logarithm of base 10 of each value in `numbers`. You should use the function `map` to achieve this.

For example:

```
[In]: get_logs10([1, 2, 3])
[Out]: [0.0, 0.3010299956639812, 0.47712125471966244]
```

Assume that:

- the length of `numbers` is non-empty and at most 10^5
- each N in `numbers` is an integer within]0, 10^5]

Hint: You can use the module `math` to compute the logarithm

In [10]:
```
# É aberta a lista 'lista' e de seguida importado o módulo 'math'.
# A lógica depois do 'return' é a seguinte: A função map() aplica determinada f
# A função que neste caso se aplica é a anónima lambda: que aplica o logaritmo
# Por fim, aplica-se a função list() para o output da 'map' function ser em lis

def get_logs10(numbers):
    lista = []
    import math
    return list(map(lambda x: math.log(x, 10), numbers))
```

```
get_logs10([1, 2, 3])
```

Out[10]:
```
[0.0, 0.30102999566398114, 0.47712125471966244]
```

Q6 [0.5p]: Create the function:

```
def greatness_evaluator(numbers, cap)
```

that, given a list of integer `numbers` and an integer `cap` returns the number of elements in `numbers` that are greater than `cap`. You should use the function `filter()` to achieve this.

For example:

```
[In]: greatness_evaluator(numbers=[50, 200, 300], cap=100)
[Out]: 2
```

Assume that:

- length of `numbers` > 0
- each N in `numbers` is an integer within [-inf, inf]
- `cap` is an integer within [-inf, inf]

In [11]:
```
# Segue praticamente a mesma lógica da função anterior, só que com mais ajuda c
# Cientificamente, seguimos a lógica "do geral para o particular". Esta explica
# A função lambda tem 1 argumento: numbers[x] > cap.
# A função filter() filtra o input com de acordo com a condição definida pela f
# A função list transforma o resultado da função filter em formato lista.
# A função len devolve o nº de elementos da lista definida pela função list que


def greatness_evaluator(numbers, cap):
        return len(list(filter(lambda x: x > cap, numbers)))

greatness_evaluator(numbers=[50, 200, 300], cap=199)
```

Out[11]:
```
2
```

Q7 [0.5p]: Create the function:

```
def get_stats(numbers)
```

that, given a list of `numbers` should return a tuple with the mean and standard-deviation value of the population of said list.

For example:

```
[In]: get_stats([1, 10, 5.5, 1.2, -4.2])
[Out]: (2.7, 4.7724207693789955)
```

Assume that:

- the length of `numbers` is non-empty and at most 10^5
- each N in `numbers` is an integer or float within [-10^5, 10^5]
- **you use the std formula for the population (N) and not the sample (N-1)**

In [12]:
```
# Nesta aprendi que não devo ignorar se encontra mais acentuado no enunciado, r
# Não para a sample. O 'p'  em "pstdev" custou-me mais de 2h de vida...Sad face
# É importado o módulo statistics e empregue para realizar a média e o desvio p

def get_stats(numbers):
    import statistics
    mean = statistics.mean(numbers)
    std = statistics.pstdev(numbers) # este p antes de stdev.(numbers)
    result = (mean, std)
    return result

get_stats([1, 10, 5.5, 1.2, -4.2])
```

Out[12]:
```
(2.7, 4.7724207693789955)
```

Q8 [1.0p]: Create the function:

```
def max_one_digit(integers)
```

that, given a list `integers` consisting of N integers, returns the maximum among all one-digit integers.

For example:

```
[In]: max_one_digit([-6, -91, 1011, -100, 84, -22, 0, 1, 473])
[Out]: 1
```

Assume that:

- the length of `integers` is non-empty and at most 10^5
- each N in `integers` is an integer within [-10^5, 10^5]
- there is at least one element in the list with 1 digit

In [13]:
```
# É criada a var 'f' e é-lhe atribuída a função filter(). Esta última vai filtr
# "Maiores ou iguais a -10 e menores que 10". Aseguir ao 'return' a função list
# de uma lista e a função max() para incluir somente na lista o maior inteiro.

def max_one_digit(integers):
    f = filter(lambda x: x >= -10 and x < 10, integers)
    return max(list(f))
```

```
max_one_digit([-5, -6, -7, 3213, 4324, 0, 1, 2])
```

Out[13]:
```
2
```

Q9 [1.0p]: Create the function:

```
def count_digits(numbers, start, stop)
```

that counts the number of digits in a subset of a list, given the following arguments:

- `numbers` : list of integer numbers
- `start` : start index of the subset (inclusive).
- `stop` : final index of the subset (inclusive)

For example:

```
[In]: count_digits(numbers=[10, -2, 0, 5, 20, 15, 18, 1],
start=2, stop=4)
[Out]: 4
```

The function should output the following:

The function starts by selecting a subset of the list between the seconds and fourth indexes (`[0, 5, 20]`) and then sums the number of digits in the subset (`1 + 1 + 2`) which results in `4` .

Assume that:

- the length of `numbers` is non-empty and at most 10^5
- each N in `numbers` is an integer within [-10^5, 10^5]
- the `start` is non-empty integer within [0, 10^5]
- the `stop` is non-empty integer within [0, 10^5]
- `stop` > `start` stop
- `stop` <= `len(numbers)-1`

In [14]:
```
def count_digits(numbers, start, stop):
    x = numbers[start:(stop+1)]
    return sum(len(str(y)) for y in x)

count_digits(numbers=[10, -2, 0, 5, 20, 15, 18, 1], start=2, stop=4)
```

Out[14]:
```
4
```

Q10 [1.0p]: Create the function:

```
def xsort(numbers)
```

that, given a list of integer `numbers` returns a new list, following these rules:

1. Sorts from higher to lower (desc)
2. Sums each element with the previous one
3. Returns the new list

For example:

```
[In]: xsort([2, 3, 1, 5])
[Out]: [5, 8, 5, 3]
```

1. In this case, we start by sorting the input (desc): `[5, 3, 2, 1]`
2. Sum N with N-1: `[5+0, 3+5, 2+3, 1+2]`. Notice that you should not sum anything to the first element.
3. Return the new list: `[5, 8, 5, 3]`

Assume that:

- the length of `numbers` is non-empty and at most 10^5
- each N in `numbers` is an integer within [-10^5, 10^5]

In [15]:
```
# O porquê do comentário em baixo: depois de MUITAS tentativas sem sucesso de c
# pela malta no stackoverflow... o seguinte código foi o único que resultou.
# Depois de um fds inteiro a resolver problemas em python, considerei a melhor
# Um (próximo) dia espero compreender. Quem deixou esse código disse "If you do
# bit esoteric"... thats the reason for those words.

def xsort(numbers):
    numbers.sort(reverse = True)
    b=0
    lista = [b + (b := a) for a in numbers] # I didn't mind getting a little bi
    return lista

xsort([2, 3, 1, 5])
```

Out[15]:
```
[5, 8, 5, 3]
```

Q11 [1.0p]: Create the function:

```
def difference_in_days(start_date, end_date)
```

that, given a string `start_date` and a string `end_date`, computes and returns an integer that is the difference in days between the two dates. For this exercise, we will include the end date in calculation (add 1 day).
Note that the inputs are both strings in format `year-month-day`, e.g. `2023-01-25` is 25 January 2023.

For example:

```
[In]: difference_in_days("2023-01-01", "2023-01-05")
[Out]: 5
```

In the previous example, 5 days elapsed between the January 5 (inclusive) and January 1 (inclusive).
Let's take a look at a different example:

```
[In]: difference_in_days("2023-02-10", "2023-03-10")
[Out]: 29
```

Assume that:

- both `start_date` and `end_date` are non-empty strings
- the `end_date` is always after or equal to the `start_date`
- both dates are in format `year-month-day`, e.g. `2023-01-25`
- the year in both strings has 4 digits
- the month in both strings has 2 digits
- the day in both strings has 2 digits

In [16]:
```
# Esta já tinha feito na W3Resource. Importar o módulo date. Dar slipt nas stri
# Transformar as strings em inteiros e atribui-los 2 var, que são subtraídas, c
# as duas datas do input. O '+1' é para incluir a end date no cálculo.

def difference_in_days(start_date, end_date):

    from datetime import date

    year_s, month_s, day_s = start_date.split("-")
    year_e, month_e, day_e = end_date.split("-")

    date_s = date(int(year_s), int(month_s), int(day_s))
    date_e = date(int(year_e), int(month_e), int(day_e))

    result = date_e - date_s
    days = result.days + 1
    print(days)

difference_in_days("2023-02-10", "2023-03-10")
```

```
29
```

Q12 [1.5p]: Create the function:

```
def common_chars(str1, str2)
```

that, given two strings `str1` and `str2`, returns a new string that contains all the characters that appear in both strings. The returned string should not contain duplicate characters, and the characters should appear in the same order as they appear in the first string.

For example:

```
[In]: common_chars("hello", "world")
[Out]: "lo"
```

In this case, the characters that appear in both "hello" and "world" are "l" and "o". The returned string should not contain duplicate characters, so the final result is "lo".

Be aware that strings may have varied lengths and, or cases, for example:

```
[In]: common_chars("I'm 10 years old, how about you?", "I'm
12!")
[Out]: "I'm 1"
```

Assume that:

- `str1` and `str2` are non-empty strings
- the characters in `str1` and `str2` are ASCII characters (i.e., their ordinal values are between 0 and 127). That includes numbers and punctuation, for example
- uppercase and lowercase characters account for different chars. Char `A` is different than `a`, for example
- the strings may have no chars in common

In [17]:
```python
# É atribuída à var 'common' uma open string. É feita uma iteração pela str1. I
# na str2. A condição definida é "Se o i (elemento da 1º iteração-str1) for igu
# e não estiver na var 'common' acrescenta-o à var 'common'". Dá return da var,
# que apareciam nas duas strings do input.

def common_chars(str1, str2):
    common = ""
    for i in str1:
        for x in str2:
            if i==x and i not in common:
                common += i
    return common

common_chars("HELLO", "WORLD")
```

Out[17]:
```
'LO'
```

Q13 [1.5p]: Create the function:

```
def find_duplicates(nums)
```

that, given a list of integers `nums`, returns a list of all integers that appear more than once in the list, sorted from smallest to highest.

For example:

```
[In]: find_duplicates([4, 3, 2, 7, 8, 2, 3, 1])
[Out]: [2, 3]
```

In this case, the integers 2 and 3 appear more than once in the list, so they are included in the output list.

Assume that:

- the length of `nums` is non-empty and at most 10^5
- each N in `nums` is an integer within [-10^5, 10^5]

In [18]:
```
def find_duplicates(nums):
    seen = set()
    seen_add = seen.add
    seen_twice = set(x for x in nums if x in seen or seen_add(x))
    return list(seen_twice)

find_duplicates([4, 3, 2, 7, 8, 2, 3, 1])
```

Out[18]:
```
[2, 3]
```

Part II - K-Means from scratch! [10.0p]

K-Means is one of the most popular "clustering" algorithms, which you perhaps are already familiar with from Data Mining or Machine Learning curricular units. K-means estimates the location of k centroids that represent the typical value in each cluster. In that sense, an observation is considered to be part of a particular cluster if it is closer to that cluster's centroid than to any other centroid.

The K-means algorithm is iterative and can be summarized as follows:

1. Start by choosing value K (number of clusters) that defines the number of clusters;
2. Initialize the locations of the centroids, which we can do by randomly selecting K points from your dataset as initial locations of the centroids;
3. Calculate the distance of all other points to each of the K centroids;
4. Associate each point to the cluster of the closest centroid;
5. Update the centroid position, by computing the average coordinates of all points assigned to each cluster;
6. Evaluate the average change in the positions of the centroids, as a measure of convergence (we assume the algorithm converged to the solution when the positions of the centroids don't change more than a given tolerance threshold);
7. Repeat steps 3-6 until either the centroids no longer move more than a tolerance threshold or until you repeated these steps at least for a specified number of iterations (niter)

The algorithm is thus parameterized by the number of clusters (K), the maximum number of iterations (niter), and the tolerance threshold (tol).

Important keywords:

- **points**: data point coordinates `(e.g.: [[0, 0.9], [0.3, 0.5], [0.8, 0.75], [0.1, 0.1]])`
- **centroids**: center of each cluster `(e.g.: [[0.5, 0.5], [0.8, 0.9]])`
- **clusters**: cluster index of each point `(e.g.: [0, 0, 1, 0])`

Relevant notes:
We must devise a strategy to test our code at each step, to assess if it performs as expected. To that end, we have split each step of the algorithm into different blocks, and we ask you to test each block against a simple scenario where the outcome is predictable and can be computed by hand. In some steps, however, we want to pass a more comprehensive dataset with similar properties to the real dataset that we will be working with.

Such, test datasets try to create realistic conditions in which your algorithm is expected to work correctly, and for which we can also assess if the solution matches our expectations. This allows us to have a general understanding of the algorithm performs as expected in "real-world" conditions.

Get data

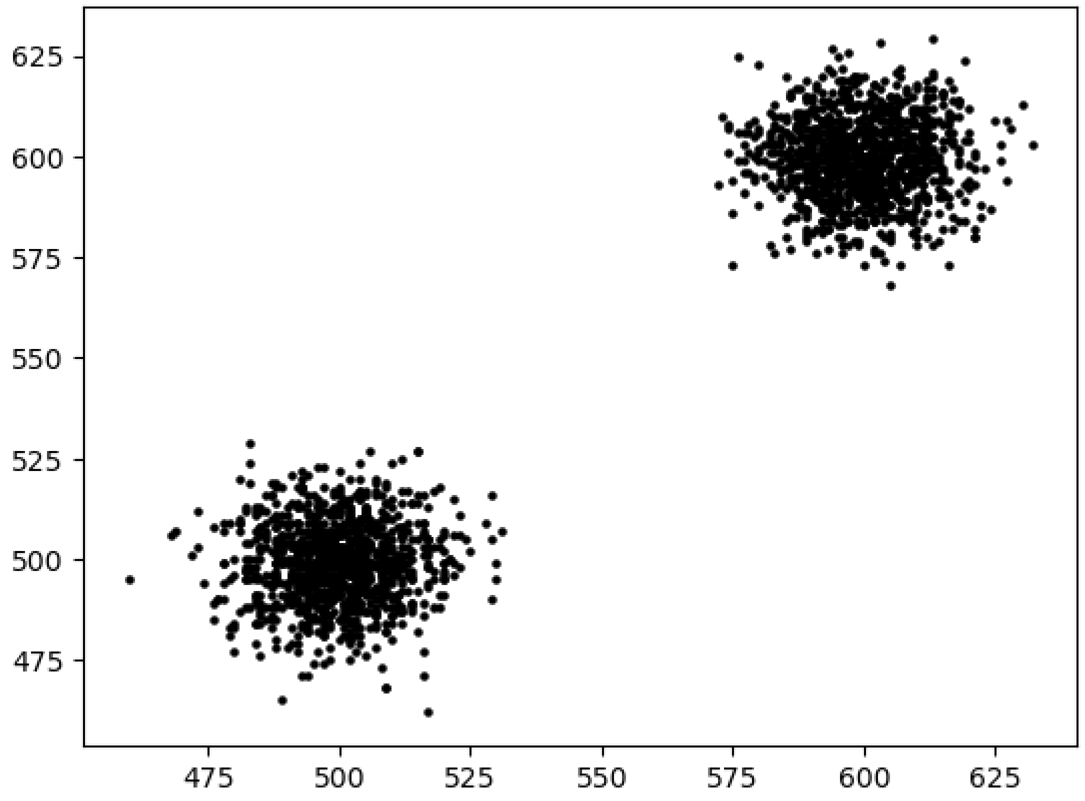
For this assignment, we will resort to a dataset that contains two distinct clusters. We generated the clusters by sampling points from two distinct gaussian distributions with different averages and the same standard deviation.
As such, our algorithm should be able to identify each cluster easily and place the centroids close to the averages of the two distributions.

Run the cell below to download, and visualize your Test dataset.
We will use this dataset to evaluate your code.

In [19]:
```python
import pandas as pd
import matplotlib.pyplot as plt

test_data = pd.read_csv(
    "https://www.dropbox.com/s/gax1l68jsarxqt9/data_test.txt?dl=1",
    header=None
).values.tolist()
```

In [20]:
```python
# Visualize
plt.scatter(
    x=list(map(lambda x: x[0], test_data)),
    y=list(map(lambda x: x[1], test_data)),
    color ='k',
    s=5
)
plt.show()
```



Step 1: Seeds (initial centroids) [1.0p]

Write the function:

```
def get_seeds(points, k)
```

that, given a list of points `points`, and the number of centroids `k`, returns a list with `k` elements **sampled randomly and without replacement** from `points`. Without replacement means that the same point cannot be picked twice.

For example:

```
[in]: get_seeds(points=[[1, 1], [2, 2], [3, 3], [4, 4], [5,
5]], k=2)
[out]: [[1, 1], [5, 5]]
```

In [21]:
```
# A função random.sample(seq, n) gera N samples únicas a partir de uma sequênci
# lis, set, string, tuple.

def get_seeds(points, k):
    import random
    x = random.sample(points, k)
    return x

get_seeds(points=[[1, 1], [2, 2], [3, 3], [4, 4], [5, 5]], k=2)
```

Out[21]:
```
[[1, 1], [5, 5]]
```

Step 2: Distance [1.5p]

Create the function:

```
def euclidean_distance(p1, p2)
```

That, given two points pf coordinates `p1` and `p2`, computes the euclidean distance between them.

Recall that the euclidian distance (d) between two points (`p1` and `p2`) in a 2-dimensional space is given by:

$$d(p1, p2) = \sqrt{(x1 - x2)^2 + (y1 - y2)^2} \qquad (2)$$

where $p_1 = (x1, y1)$ and $p2 = (x2, y2)$.

For example:

```
[In]: euclidean_distance(p1=[1, 3], p2=[7, 2])
[Out]: 6.082762530298219
```

BONUS POINTS [0.5p]

- Skip it if you don't know or don't have time
- The bonus points (0.5) will only stack to your grade if your HW grade < 20.

Adapt your `euclidean_distance` function to compute the euclidean distance between two points of arbitrary dimension. In general, for arbitrary dimensionality, the square of the distance can be computed as:

$$d(p1, p2) = \sqrt{\sum_{i=0}^{D} (xi - yi)^2} \qquad (3)$$

where $p1 = (x1, x2, \ldots, x_{D-1}, x_D)$ and $p_2 = (y1, y2, \ldots, y_{D-1}, y_D)$.

For example:

```
[In]: euclidean_distance_complex(p1=[1, 3, -9, 12], p2=[7, 2,
0, 5])
[Out]: 12.922847983320086
```

In [22]:
```
# Euclidean distance calculated with 'math' module

def euclidean_distance(p1, p2):
    import math
    print(math.dist(p1, p2))
```

```
euclidean_distance(p1=[1, 3], p2=[7, 2])
```

```
6.0827625302982185
```

In [23]:
```
# Euclidean distance complex calculated with 'math' module but with the zip fu
# The zip() iterates the operation before the loop through p1 and p2.
# Then, we're calculating the sqrt root of sum_squares

def euclidean_distance_complex(p1, p2):
    import math
    sum_squares = sum((p1 - p2)**2 for p1, p2 in zip(p1, p2))
    x = math.sqrt(sum_squares)
    print(x)

euclidean_distance_complex(p1=[1, 3, -9, 12], p2=[7, 2, 0, 5])
```

```
12.922847983320086
```

Step 3 Centroids [2.0p]

Write the function:

```
def get_centroids(points, clusters, k)
```

That, given the following arguments:

- `points` : List of lists, where each list is a set of coordinates for each row (1 point per row).
- `clusters` : a 1-dimensional list that indicates the cluster to which each point is associated. returns a list with the mean coordinates of the points associated with each cluster. This will represent the centroid of each cluster. The returned list should have the same number of columns as points, and rows as the length of k.

For example:

```
[In]: points = [[0.5, 0.6], [0.8, 0.4], [1.0, 1.0]]  # 3 sets
of points
[In]: clusters = [0, 0, 1]  # cluster 0, cluster 0, and cluster
1
[In]: k = 2  # 2 clusters
[In]: get_centroids(points, clusters, k)

[Out]: [[0.65, 0.5], [1.  , 1.  ]]
```

In this example, the centroid of cluster 0 is computed as: [(0.5 + 0.8)/2, (0.6+0.4)/2] = [0.65, 0.5].
In the case of cluster 1: [1, 1]

Assume that:

- the values in `clusters` are integers and within [0, k-1]

In [24]:
```
def get_centroids(points, clusters, k):

# Given a list of points, respective clusters(0,k-1) and number of clusters, re

    centroids = []
    for i in range(k):                      # Iterates over each cluster

        cluster_points = []                 # The cluster points will come organize

        for x in range(len(points)):

            if clusters[x] == i:               # organize the points per clusters w
                cluster_points.append(points[x])

        centroid_x = 0                               # Iniciates centroid coordir
        centroid_y = 0
        for y in cluster_points:                     # Organize x and y coordinat
            centroid_x += y[0]
            centroid_y += y[1]

        if len(cluster_points) > 0:                # If the cluster_points list

            centroids.append([                       # Calculates each centroid p
                (centroid_x/len(cluster_points)),
                (centroid_y/len(cluster_points))])

    return centroids


get_centroids([[0.5, 0.6], [0.8, 0.4], [1.0, 1.0]], [0, 0, 1], 2)
```

Out[24]:
```
[[0.65, 0.5], [1.0, 1.0]]
```

Step 4: Clusters [2.0p]

Write the function:

```
def get_clusters(points, centroids)
```

That, given a list of `points` and a list of `centroids`, returns a one-dimensional list that indicates the index of the closest centroid to each point. To determine the distance between a point and a centroid, you can use your euclidean distance function created earlier **(step 2)**.

You will need to compute the distance between each point to each centroid. Different approaches can be taken to do this. *(e.g. you can start by computing a d by k distance matrix matrix between each of the d points and the k centroids.)*

For example:

```
[In]: get_clusters(points=[[1,1], [2, 2], [3, 3], [4, 4], [5,
5]], centroids=[[1.5, 1.5], [4.5, 4.5]])
[Out]: [0, 0, 0, 1, 1]
```

In this example, we have 5-point coordinates and two clusters (2 centroids).
For the first point, the distance between `[1, 1]` and centroid 0 (`[1.5, 1.5]`) is ~0.71 while the distance to centroid 1 (`[4.5, 4.5]`) is ~4.95. Therefore, point `[1, 1]` is closer to centroid 0, so its assigned cluster will be cluster 0.
The same logic is applied to the remaining points.

In [25]:
```
def get_clusters(points, centroids):

    centroid_index = []
    for i in range(len(points)):                    # Nested loop: alocate to distanc
        distances = []
        for x in range(len(centroids)):
            import math
            d = math.dist(points[i], centroids[x])        # Euclidean distance c
            distances.append(d)

        closest_centroid_index = distances.index(min(distances))  # this line i
        centroid_index.append(closest_centroid_index)

    return centroid_index

get_clusters([[1,1], [2, 2], [3, 3], [4, 4], [5, 5]], [[1.5, 1.5], [4.5, 4.5]])
```

Out[25]:
```
[0, 0, 0, 1, 1]
```

Step 5: Error [1.5p]

Write the function:

```
def get_error(old_centroids, new_centroids)
```

That takes two input arguments:

- `old_centroids` : list with centroid positions from the previous iteration (iteration N-1).
- `new_centroids` : list with new computed centroid positions from the current iteration (iteration N).

The function should return the average euclidean distance between the old and new positions of each centroid.

This will give us a measure of whether our algorithm is still converging to the solution or got too good for approximation.

We will use a tolerance threshold in the average change in the position of the centroids to make this decision.

To get new centroids, you must use the `get_centroids` function (from Step 3) and apply it to the `get_clusters` (from Step 4)

Test your function using the `centroids` obtained in Step 3 as `old_centroids`, and the function `get_centroids` to calculate new centers for the `new_centroids` argument.

For example:

```
[In]: get_error(old_centroids=[[0.5, 0.3], [0.6, 0.7]],
new_centroids=[[0.55, 0.5], [0.7, 0.7]])
[Out]: 0.15307764064044152
```

In this example, the old centroids of clusters 1 and 2 are, respectively, [0.5, 0.3] and [0.6, 0.7]. The new centroids of clusters 1 and 2, respectively, are [0.55, 0.5] and [0.7, 0.7]. This means the error/distance between the old and new centroids for cluster 1 is ~0.2061 and for cluster 2 is ~0.0999. The average of these values is 0.153.

In [26]:
```
def get_error(old_centroids, new_centroids):
    import math

    total_distance = 0                      # list used to store the sum of the dis
    num_centroids = len(old_centroids)  # stores the number of centroids
    for i in range(num_centroids):

        total_distance += math.dist(old_centroids[i], new_centroids[i])  # calc

    avg = total_distance / num_centroids
    return avg

get_error([[0.5, 0.3], [0.6, 0.7]], [[0.55, 0.5], [0.7, 0.7]])
```

Out[26]:
```
0.1530776406404415
```

Step 6: Wrap it up [2.0p]

You're almost there! Wrap everything up in a function called:

```
def kmeans(points, k, n_iter, tol)
```

That takes the following arguments:

- `points` : list of point coordinates
- `k` : number of clusters, **default should be 2**
- `n_iter` : number of iterations to test, **default should be 100**
- `tol` : error tolerance threshold below which your algorithm should stop, **default should be 0.00005**

The function should return a tuple of two elements: `(clusters, location of centroids sorted)`. Be aware that it is expected that you change the default arguments of the function, as asked for in the arguments specification.

You should use the previously created functions to achieve `kmeans`. The `n_iter` and `tol` arguments should serve as stoppers for your algorithm, this is, for example, if `n_iter=10` then your algorithm should be working during 10 iterations, trying to find the optimal cluster centroids. If `tol=0.005` then it means that, if the average error/distance between the old and new centroids is equal to or smaller than your defined tolerance threshold, the algorithm should stop.
In sum, the stopping criteria are as follows:

- Your algorithm should stop if the error between old centroids and new centroids <= `tol`
- Your algorithm should stop after `n_iter` interactions

For example:

```
[In]: kmeans(points=[[0.5, 0.5], [1, 1], [2, 2],[3, 3]], k=2,
n_iter=10, tol=0.1)
[Out]: ([0, 0, 1, 1], [[0.75, 0.75], [2.5, 2.5]])
```

In this example, my final clusters are [0, 0, 1, 1], which state that:

- points `[0.5, 0.5]` and `[1, 1]` belong to cluster 0
- points `[2, 2]` and `[3, 3]` belong to cluster 1
- the centroid of cluster 0 is `[0.75, 0.75]`
- the centroid of cluster 1 is `[2.5, 2.5]`

NOTE!: A common issue with K-Means is that empty clusters may be created, e.g., if there are 3 clusters and all points are closer to cluster 0 or cluster 1. In these cases, your code could raise errors when computing statistics such as cluster 2 centroids (which is empty). However, with the provided data, this should not happen and you don't need to implement anything to overcome this problem. If such a case arises, we recommend you re-run your code. **All clusters are expected to have, at least, 1 observation**.

Your `kmeans` function will be evaluated against the `test_data`, therefore, you should check that your code works against this data.

```
In [27]: def kmeans(points, k=2, n_iter=100, tol=0.00005):
             import math
             error = float('inf')                # initializes the error variable to
             centroids = get_seeds(points, k)    # get_seeds function generate to th

             while n_iter > 0 and error > tol:

                 old_centroids = centroids           # Those centroid points are now the

                 clusters = get_clusters(points, centroids)    # call the get_clusters f

                 centroids = get_centroids(points, clusters, k) # call the get_centroids

                 new_centroids = centroids
```

```
        error = get_error(old_centroids, new_centroids)
        n_iter -=1
    return (clusters, sorted(centroids))
```

Visualize clusters

This segment allows you to visualize your work by visually displaying how your clustering algorithm `kmeans` was able to group observations into similar groups.

It will not be taken into account for grading, however, it is not expected that errors will be raised. If they do so, a penalty will be applied. If you are having trouble, just leave this with the provided examples that work.

The function `plot_clusters` takes three arguments:

- `points` : list of point coordinates
- `clusters` : 1-D list of assigned clusters (length = length of points)
- `centroids` : ordered centroids of each cluster

The code will plot all the points and then the respective cluster centroids with random colors.

In [28]:
```
def plot_clusters(points, clusters, centroids):


    r = lambda: random.randint(0,255)
    color_dict = {}
    for cluster in range(len(centroids)):
        color_dict[cluster] = str("#%02X%02X%02X" % (r(),r(),r()))

    # Colors list
    colors = list(map(lambda c: color_dict.get(c), clusters))

    # Plot points
    points_in_cluster_indexes = list(filter(lambda c: c == cluster, clusters))
    points_in_cluster = []
    plt.scatter(
        x=list(map(lambda x: x[0], points)),
        y=list(map(lambda x: x[1], points)),
        s=30,
        color=colors
    )

    # Plot centroids
    plt.scatter(
        x=list(map(lambda x: x[0], centroids)),
        y=list(map(lambda x: x[1], centroids)),
        s=100,
        marker="x",
        linewidths=5,
        color="black"
    )
    plt.show()
```

Here's an example of how the plot works.

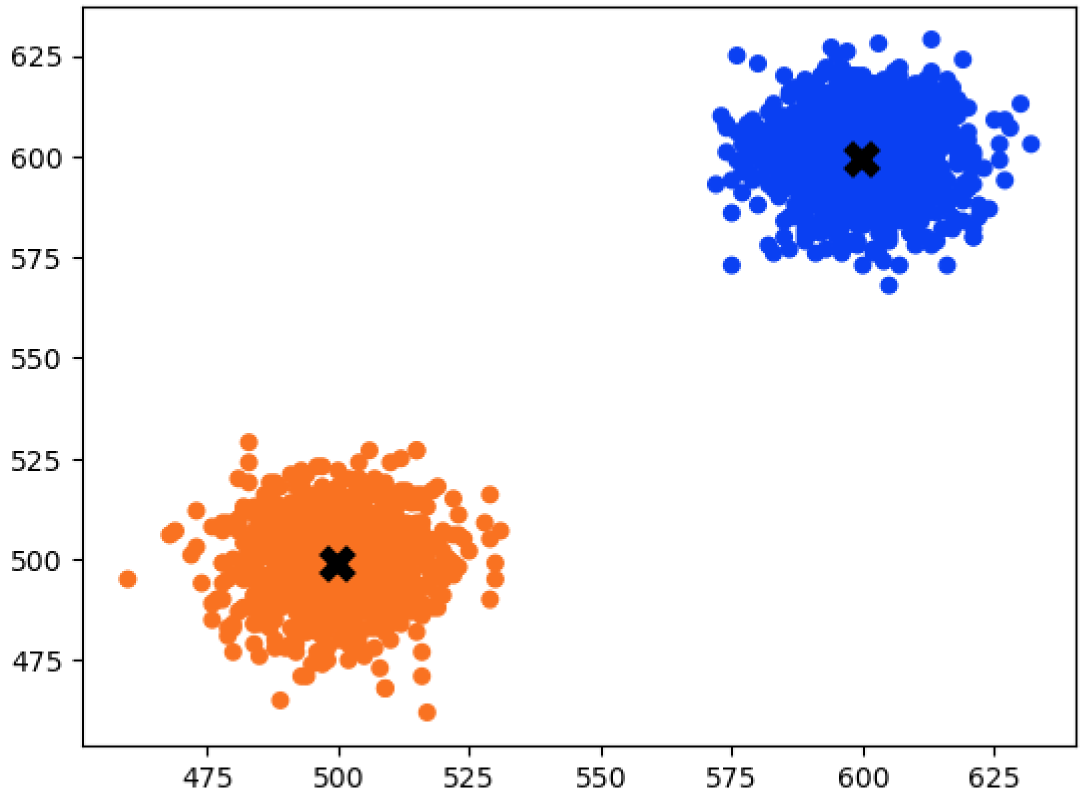
```
In [ ]: test_visualization_points = [[0.89, 0.27], [0.34, 0.36], [0.33, 0.19], [0.67, (
        test_visualization_clusters = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
        test_visualization_centroids = [[0.592, 0.278], [0.464, 0.768]]

        plot_clusters(
            points=test_visualization_points
            clusters=test_visualization_clusters
            centroids=test_visualization_centroids
        )
```

Test your `kmeans` function by calling it against the `test_data` and saving the returned clusters and centroids. Check how the clusters behave. You should have two very distinct groups.

```
In [30]: points= test_data
         clusters = kmeans(points,2)[0]
         centroids = kmeans(points,2)[1]
         plot_clusters(points,clusters,centroids)

         counter0 = 0
         counter1 = 0
         for i in kmeans(points,2)[0]:
             if i == 0:
                 counter0 += 1
             if i == 1:
                 counter1 +=1
         print(counter0,counter1)
```



```
1024 1023
```

In [ ]:

```

```