Python for Data Sciences

Course and exam presentation, Data Managing

- Armando Camerlingo

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

- Armando Camerlingo
- PhD @ Unitus
- HPC/AI Specialist @ Prisma S.p.A

ITINERIS

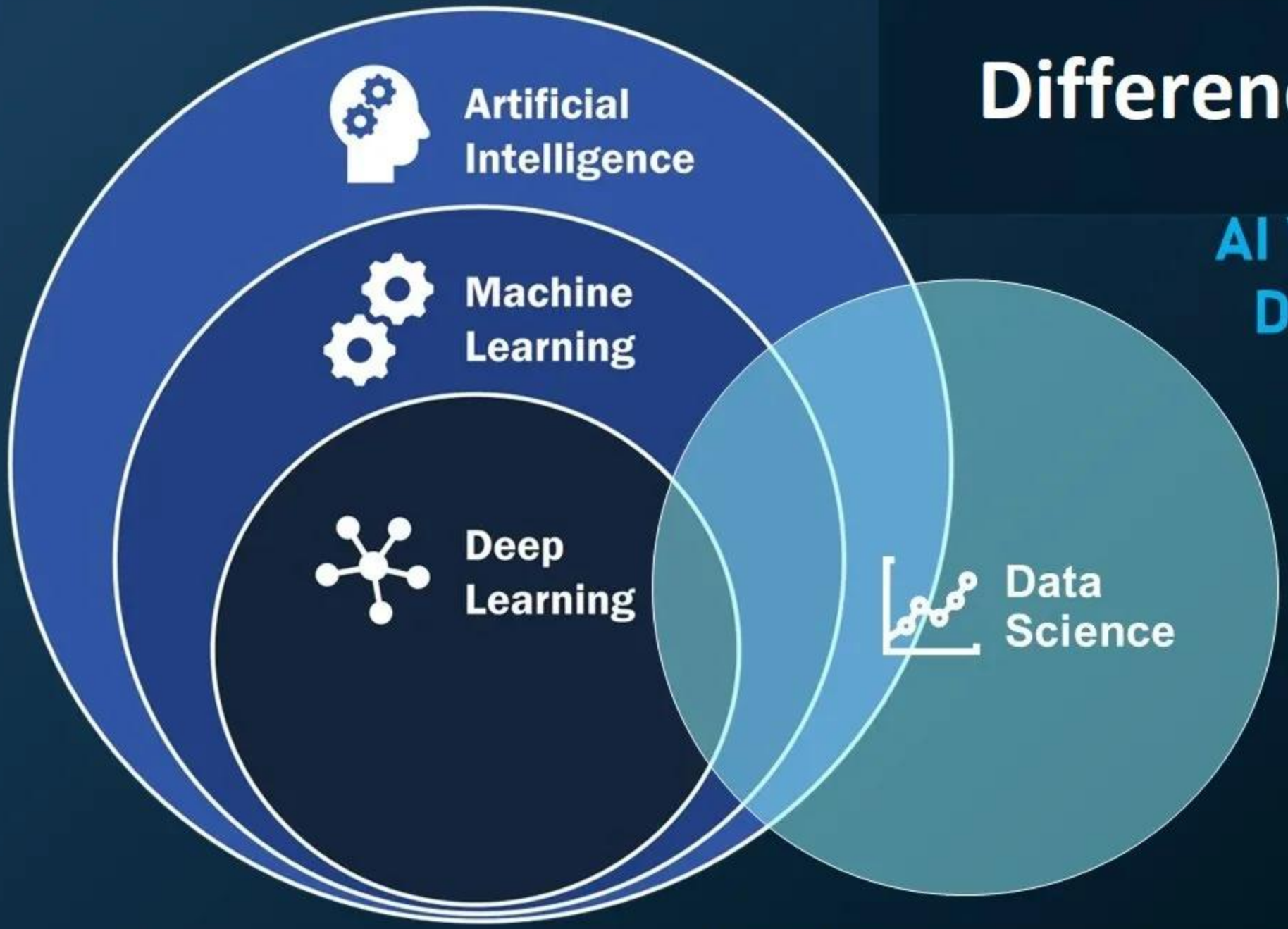
PRISMA

- Mail address: armando.camerlingo@unitus.it

Google CoLab

Difference Between

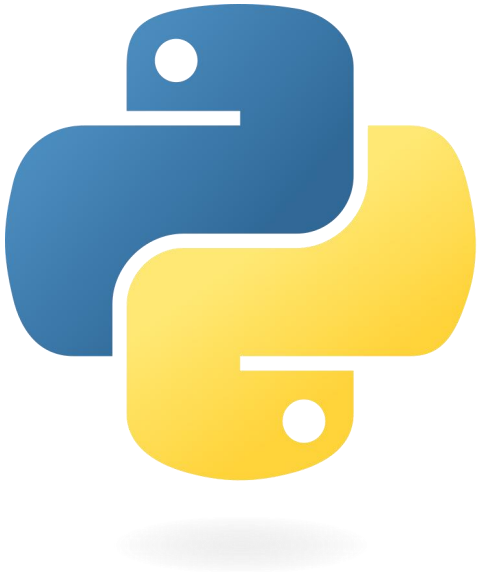
AI VS ML VS DL VS DS



https://medium.com/@shahinaperveen9444/differences-between-ai-ml-dl-ds-cc899d6921bf

Why Python for ML/DL?

ITINERIS

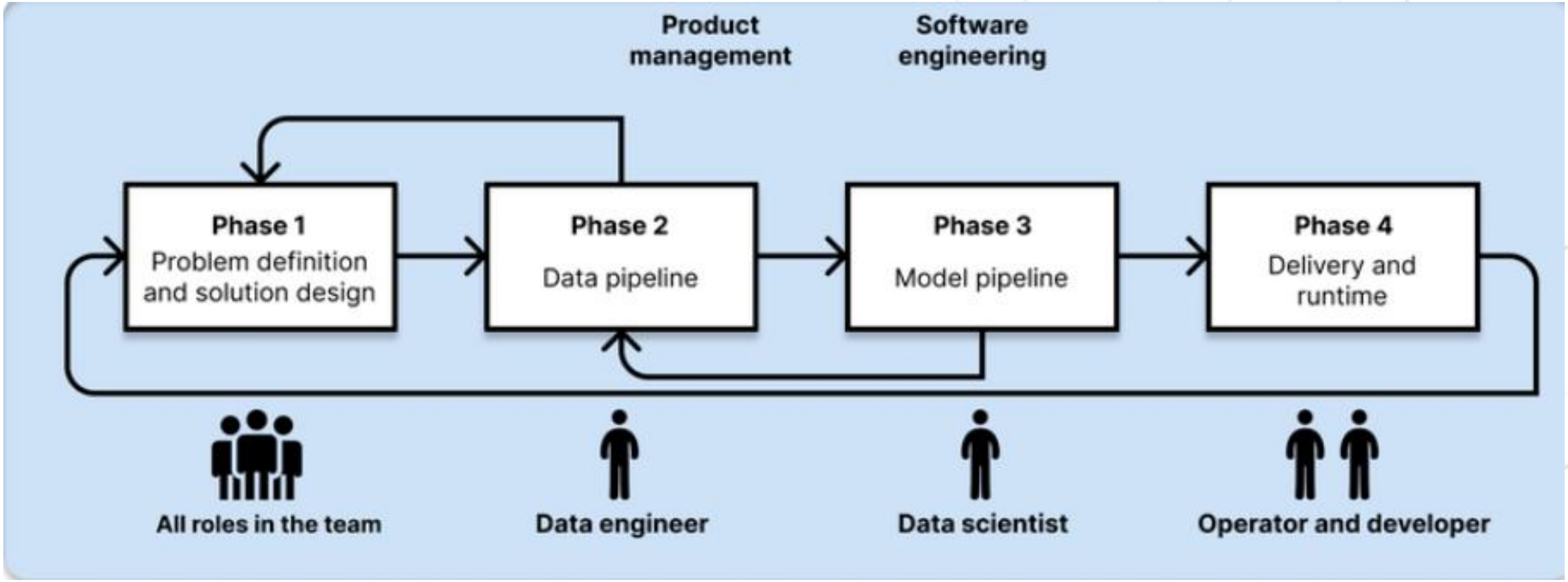


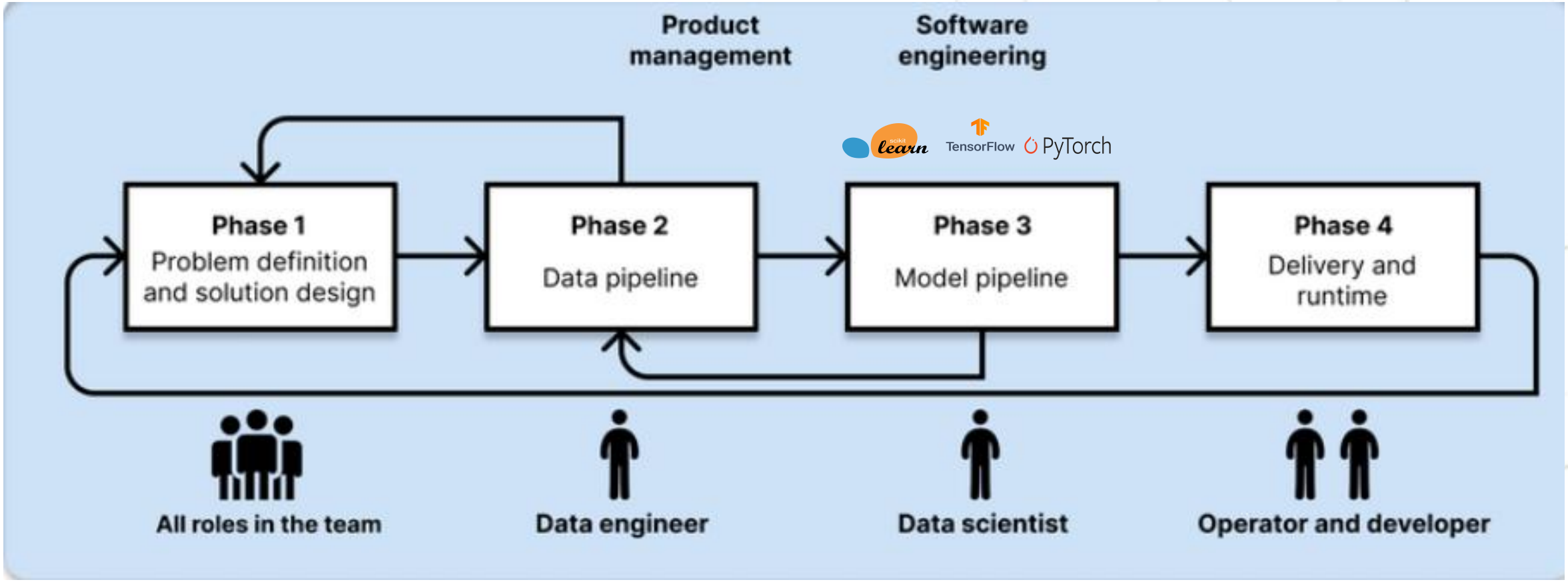
ITINERIS

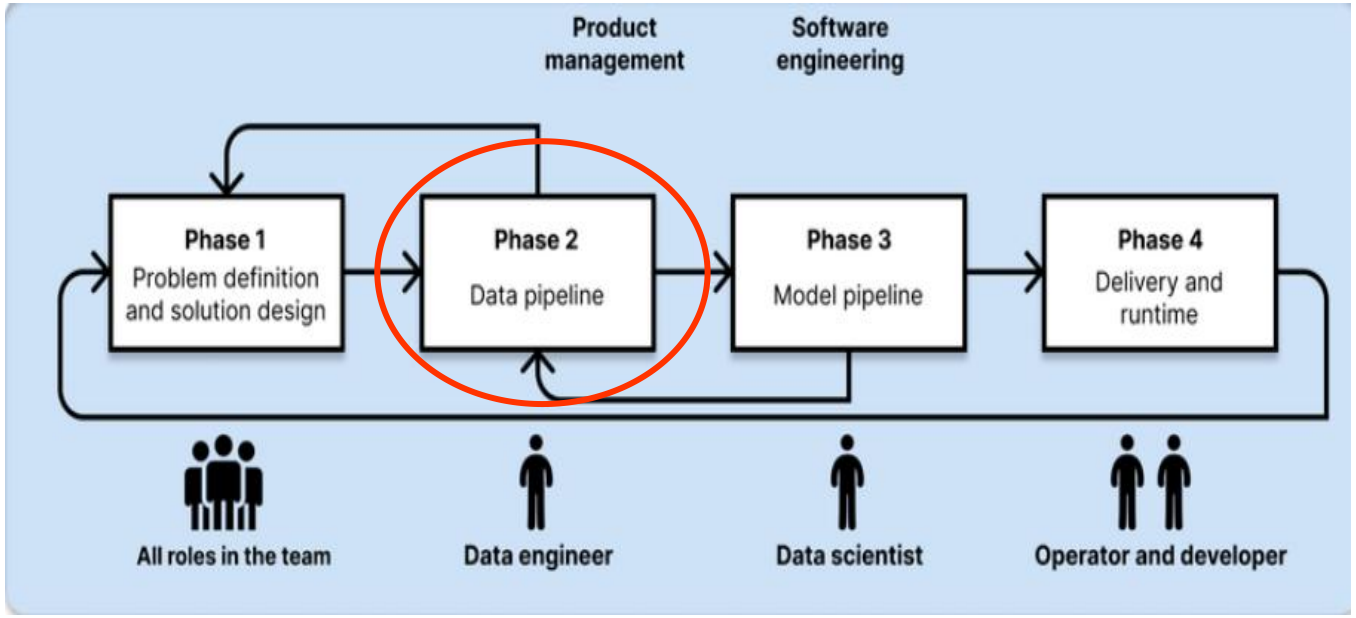
scikit learn

TensorFlow

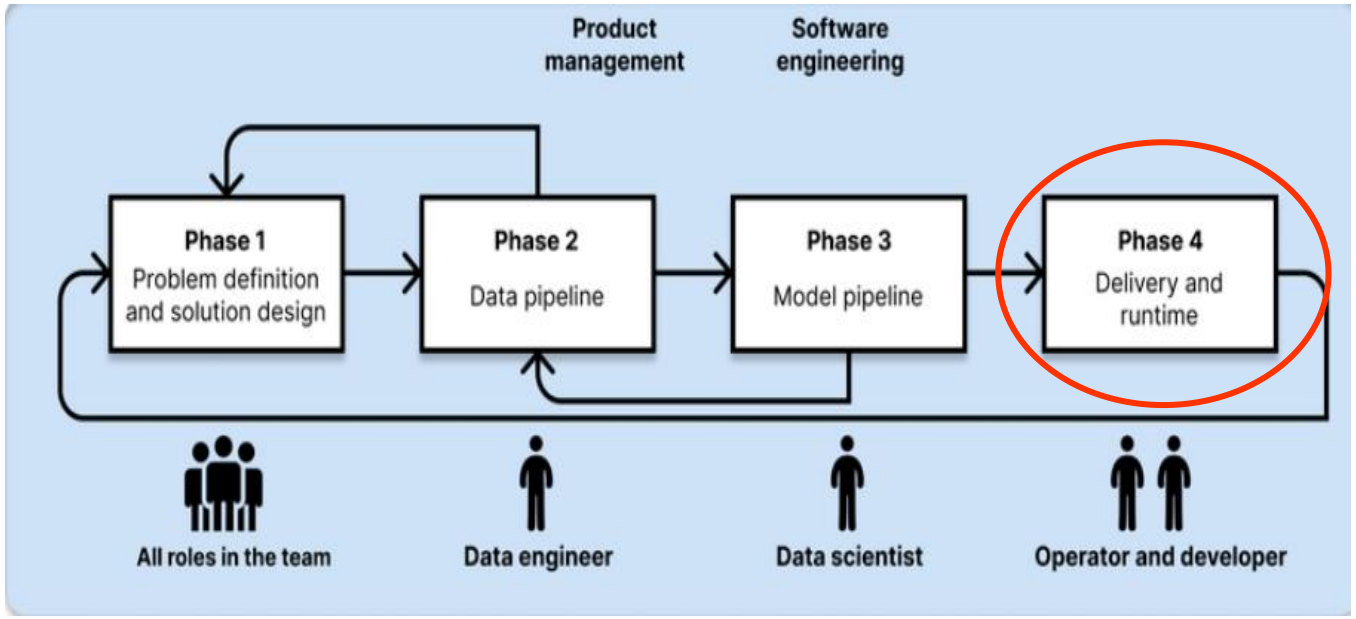
PyTorch


Product management
Software engineering
Phase 1
Problem definition and solution design
Phase 2
Data pipeline
Phase 3
Model pipeline
Phase 4
Delivery and runtime
All roles in the team
Data engineer
Data scientist
Operator and developer


Product management
Software engineering
scikit learn
TensorFlow
PyTorch
Phase 1
Problem definition and solution design
Phase 2
Data pipeline
Phase 3
Model pipeline
Phase 4
Delivery and runtime
All roles in the team
Data engineer
Data scientist
Operator and developer



- How we receive the data?
- Is the data clean?
- Is the data all relevant to our study case?
- Is the data ready to be ingested by our model?



- Where do we want to run our model?
- CPU or GPU?
- Can parallelize some parts of our code?
- Plots and metrics about our model

Course contents

- Data Managing
- Parallelization
- Data visualization
- Nvidia Libraries

Data Managing

ITINERIS

Numpy

Pandas

Polars

Numpy

NumPy (“Numerical Python”) is an open source library, used in almost every script in scientific research. NumPy works with **arrays** and it is useful when linear algebra and matrices are involved.

Numpy Array

An array is a set of values, all of the same type, and uses an index of non-negative numbers. The number of dimensions is called rank, while the shape is a tuple ot integers representing the length of the array along every dimension.

```
import numpy as np

a = np.array([1,2,3])
print(type(a))
print(a.shape)
print(a[0], a[1], a[2])
a[0] = 5
print(a)
```

Numpy Array: Attributes

NumPy’s array class is called `ndarray`. It is also known by the alias array. Here are some of their attributes:

- ndarray.dim: The number of axes
- ndarray.shape: The number of items in each axis (e.g. rows, columns)
- ndarray.size: The total number of elements
- ndarray.dtype: The datatype used in the array
- ndarray.itemsize: The size of each element in bytes
- ndarray.data: The address in memory

Numpy Array: Attributes

```
import numpy as np
a = np.arange(21).reshape(3, 7)
a
```

```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])
```

```
a.ndim
```

```
2
```

```
a.shape
```

```
(3, 7)
```

```
a.size
```

```
21
```

Numpy Array: Attributes

```
a.dtype
```

```
dtype('int64')
```

```
a.itemsize
```

```
8
```

```
a.data
```

```
<memory at 0x788dfd97f370>
```

Numpy Array: Creation

```
a = np.array([1,2,3])
b = np.array([1.,2.,3.])
a.dtype, b.dtype
```

```
(dtype('int64'), dtype('float64'))
```

```
c = np.array([[1,2,3,4],
              [4,6,7,8]])
c
```

```
array([[1, 2, 3, 4],
       [4, 6, 7, 8]])
```

```
d = np.array([[4, 6], [2, 7]], dtype = np.float32)
d
```

```
array([[4., 6.],
       [2., 7.]], dtype=float32)
```

Numpy Array: Creation

```
e = np.zeros(shape = (3,4))
e
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
f = np.zeros(shape = (5,3), dtype=np.int32)
f
```

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]], dtype=int32)
```

Try to create an `empty` array of (2,3) shape with type `complex64`

Numpy Array: Creation

The `arange` function create an array of sequential numbers. It takes the start, stop and step as arguments

```
h = np.arange(2,8,2)
h
```

```
array([2, 4, 6])
```

When we want to specify the number of elements in the array , it is best to use the `linspace` function:

```
i = np.linspace(4,6,5)
i
```

```
array([4. , 4.5, 5. , 5.5, 6. ])
```

Numpy Array: Creation

`linspace` can be used together with math functions to evaluate them on a set of points

```
k = np.linspace(0, 3 * np.pi, 50)
f_sin = np.cos(k)
f_sin
```

```
array([ 1.        ,  0.98155916,  0.92691676,  0.8380881 ,  0.71834935,
        0.57211666,  0.40478334,  0.22252093,  0.03205158, -0.1595999 ,
       -0.34536505, -0.51839257, -0.67230089, -0.80141362, -0.90096887,
       -0.96729486, -0.99794539, -0.99179001, -0.94905575, -0.8713187 ,
       -0.76144596, -0.6234898 , -0.46253829, -0.28452759, -0.09602303,
        0.09602303,  0.28452759,  0.46253829,  0.6234898 ,  0.76144596,
        0.8713187 ,  0.94905575,  0.99179001,  0.99794539,  0.96729486,
        0.90096887,  0.80141362,  0.67230089,  0.51839257,  0.34536505,
        0.1595999 , -0.03205158, -0.22252093, -0.40478334, -0.57211666,
       -0.71834935, -0.8380881 , -0.92691676, -0.98155916, -1.        ])
```

Numpy Array: Creation

A previous defined array can be used to define a new array with the same shape and datatype:

```
l = np.zeros_like(j)
l.shape, l.dtype
```

```
((50,), dtype('float64'))
```

We can also create a random populated array:

```
m = np.random.normal(loc = 3, scale = 1.0, size=(2,3))
m
```

```
array([[2.30957553, 3.5405346 , 3.45089904],
       [1.84880731, 3.23814273, 2.73781081]]))
```

Arithmetic Operations


```
a = np.arange(0,10,1)
b = np.linspace(1, 5.5,10)
```

We can perform arithmetic operations on a single array or between two array of same length:

```
a+b
```

```
a**2
```

```
b*2
```

```
a > 3
```

```
a*b
```

Matrix Multiplication

Matrix multiplication can be performed with the function `dot` or with @, which is the same as the function `matmul`

```
c = np.array([[1,2,3], [4,5,6]])
d = np.array([[1,2,3,4],[1,2,3,4],[1,2,3,4]])
```

```
c.shape, d.shape
```

```
((2, 3), (3, 4))
```

```
c.dot(d)
```

```
array([[ 6, 12, 18, 24],
       [15, 30, 45, 60]])
```

```
d.dot(c)
```

```
ValueError: shapes (3,4) and (2,3) not aligned: 4 (dim 1) != 2 (dim 0)
```

Matrix Multiplication

ITINERIS

`np.dot` not works well with higher dimension arrays. Let's see how it works:

```
e = np.ones(shape = (9, 5, 7, 4))
f = np.ones(shape = (9, 5, 4, 3))
```

`np.dot` will treat the e matrix as a (9*5*7,4) and f as a (4, 9*5*3). So the resulting matrix will have a (315,135) shape and the `np.dot` will try to rebuild the array, giving results not reliable.
Instead `matmul` considers the last two dimensions as matrices and broadcasting the other ones. So, in our case e will be considered as a set of (7,4) matrices, while f as a set of (4,3) matrices. Thus, the result will have a shape of (9,5,7,3).

```
np.dot(e, f).shape
```

```
(9, 5, 7, 9, 5, 3)
```

```
np.matmul(e, f).shape
```

```
(9, 5, 7, 3)
```

Matrix Multiplication

ITINERIS

There is no really difference in computational time

```
n = 1500
g = np.arange(n**2).reshape(n, n)
h = np.arange(n**2).reshape(n, n)
```

```
%timeit g.dot(h)
```

```
8.46 s ± 356 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit g @ h
```

```
8.97 s ± 374 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit np.matmul(g,h)
```

```
9.11 s ± 332 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Ufuncs

ITINERIS

An universal function that operates element-by-element, giving an array as output. They include mathematical functions such as sin, cos, exp... . Usually compiled in C

```
A = np.arange(100).reshape(10,10)
```

```
np.sqrt(A)
```

```
np.exp(A)
```

Array indexing and slicing

Indexing and slicing let us access and select subsets of the array values.

```
B = np.arange(0,50,2)
B
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48])
```

From index 2 to 12 (not included), select the third item (Remember that indeces starts from 0!).

```
B[2:12:3]
```

```
array([ 4, 10, 16, 22])
```

```
B[0:3]=0
B
```

```
array([ 0,  0,  0,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48])
```

Array indexing and slicing

Indices in multidimensional arrays are separated by commas

```
C = np.arange(0,1000).reshape(10,10,10)
```

For example let's take all the elements from the first axes, the first 5 from the second and the last from the third

```
C[:,0:5,-1]
```

```
array([[  9,  19,  29,  39,  49],
       [109, 119, 129, 139, 149],
       [209, 219, 229, 239, 249],
       [309, 319, 329, 339, 349],
       [409, 419, 429, 439, 449],
       [509, 519, 529, 539, 549],
       [609, 619, 629, 639, 649],
       [709, 719, 729, 739, 749],
       [809, 819, 829, 839, 849],
       [909, 919, 929, 939, 949]])
```

Exercise

Create two arrays with size (3,3,3), slice away two (2,2) matricies and multiply them.

Exercise

Create two arrays with size (3,3,3), slice away two (2,2) matricies and multiply them.

Solution

```
A = np.arange(27).reshape(3,3,3)
B = np.random.uniform(0,100,27).reshape(3,3,3)
```

```
C = A[:2,:2,1] @ B[1:3,1:3,-1]
C
```

Iteration over array elements

To perform an operation on all the elements on an array we can use the `flat` operator:

```
D = np.arange(100).reshape(10,10)
```

```
for element in D.flat:
    element += 1
    print(element)
```

```
1
2
3
4
5
6
7
8
9
10
…
```

Array: Copy and Views

ITINERIS

There are multiple ways to duplicate an array and/or its content:

- No copy: a and b are the same object

```
a = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

```
b = a
b is a
```

```
True
```

```
b[0,0]=100
a
```

```
array([[100,   1,   2,   3],
       [  4,   5,   6,   7],
       [  8,   9,  10,  11]])
```

The first element of a has changed

Array: Copy and Views

There are multiple ways to duplicate an array and/or its content:

- View: new object, but shared data.

```
c = a.view()
c is a; c.base is a
```

```
(False, True)
```

```
c = c.reshape(2,6)
print(a)
print(c)
```

```
(array([[100,   1,   2,   3],
        [  4,   5,   6,   7],
        [  8,   9,  10,  11]]),
 array([[100,   1,   2,   3,   4,   5],
        [  6,   7,   8,   9,  10,  11]]))
```

A view object can be reshaped, but the data is still the same

Array: Copy and Views

There are multiple ways to duplicate an array and/or its content:

- View: new object, but shared data.

```
c = a.view()
c is a; c.base is a
```

```
(False, True)
```

```
c[0,1]=50
print(a)
```

```
array([[100,  50,   2,   3],
       [  4,   5,   6,   7],
       [  8,   9,  10,  11]])
```

The data contained in a has been modified

Array: Copy and Views

There are multiple ways to duplicate an array and/or its content:

- Copy: new object, an complete copy of the original array and its data.

```
e = a.copy()
e
```

```
(False, True)
```

```
e is a
```

```
False
```

```
e.base is a
```

```
False
```

Array: Copy and Views

There are multiple ways to duplicate an array and/or its content:

- Copy: new object, an complete copy of the original array and its data.

```
e[0,2]=1000
e
```

```
array([[ 100,    50, 1000,     3],
       [   4,     5,     6,     7],
       [   8,     9,    10,    11]])
```

```
a
```

```
array([[100,   50,    2,    3],
       [  4,    5,    6,    7],
       [  8,    9,   10,   11]])
```

Manipulating arrays

- Joining: putting contents of two array in single one with the same number of dimensions. We use the `concatenate()` function.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
```

```
arr = np.concatenate((arr1, arr2))
print(arr)
```

```
[1 2 3 4 5 6]
```

We can use axes to decide the dimension in which the array will be joined

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
```

```
arr = np.concatenate((arr1, arr2) , axis=1)
print(arr)
```

```
[[[1 2 5 6]
 [3 4 7 8]]
```

Manipulating arrays

- Stacking: putting contents of two array in single one along a new axis. We use the `stack()` function. We can specify the axis to be used to stack (default is 0).

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
```

```
arr = np.stack((arr1, arr2) , axis=1)
print(arr)
```

```
[[1 4]
 [2 5]
 [3 6]]
```

```
arr = np.stack((arr1, arr2))
print(arr)
```

```
[[1 2 3]
 [4 5 6]]
```

There are other functions that perform the same operation for fixed axis: vstack() along columns, hstack() along rows, dstack() along depth in a 3d array

Manipulating arrays

- Splitting: reverse of join, dividing an array in multiple smaller ones. We can use the `array_split()` function.

```
arr = np.array([1, 2, 3, 4, 5, 6])
```

```
newarr = np.array_split(arr, 3)
print(newarr)
```

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

If there aren't enough elements, it will adjust by itself

```
newarr = np.array_split(arr, 4)
print(newarr)
```

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

```
print(type(newarr))
```

```
<class 'list'>
```

Manipulating arrays

- Splitting: reverse of join, dividing an array in multiple smaller ones. We can use the `array_split()` function. We can also divide array with multiple dimensions.

```
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
```

```
newarr = np.array_split(arr, 3)
print(newarr)
```

```
[array([[1, 2],
        [3, 4]]),
array([[5, 6],
        [7, 8]]),
array([[ 9, 10],
        [11, 12]])]
```

We obtain 3 array of two dimensions.

Manipulating arrays

- Splitting: reverse of join, dividing an array in multiple smaller ones. We can use the `array_split()` function. We can also specify the axis along which we split.

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]]
```

```
newarr = np.array_split(arr, 3, axis=1)
print(newarr[0])
```

```
[[ 1]
 [ 4]
 [ 7]
 [10]
 [13]
 [16]]
```

Manipulating arrays

- Sorting: order the elements in an array.

```
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

```
[0 1 2 3]
```

It works also with string, boolean variables and multi-dimensions array.

```
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

```
arr = np.array([True, False, True])
print(np.sort(arr))
```

```
[False  True  True]
```

```
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

Manipulating arrays

- Searching: find a chosen values in an array and return the indices associated. We can use the `where()` function.

```
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

```
(array([3, 5, 6]),)
```

We can use also more complex criteria

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

```
(array([1, 3, 5, 7]),)
```

We can print the array elements associated to the index in this way:

```
arr[x]
```

```
array([2, 4, 6, 8])
```

Manipulating arrays

- Searching in sorted array: find the position in which the element can be inserted in order to not disrupt the sorting. We use the `searchsorted()` function.

```
arr = np.array([6, 7, 9, 10])
x = np.searchsorted(arr, 8)
print(x)
```

```
2
```

We can also search for multiple values

```
arr = np.array([1, 3, 5, 7])
x = np.searchsorted(arr, [2, 4, 6])
print(x)
```

```
[1 2 3]
```

Manipulating arrays

- Filtering: Selecting elements from an existing array using some criteria and creating a new array from them. A boolean index list can be used.

```
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr)
```

```
[41 43]
```

Only the elements associated to the index equal to true were selected in the newarr variable

Manipulating arrays

- Filtering: Selecting elements from an existing array using some criteria and creating a new array from them. A boolean index list can be used. The Boolean index list can be created also from a set of conditions:

```
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element is higher than 42, set the value to True, otherwise False:
  if element > 42:
    filter_arr.append(True)
  else:
    filter_arr.append(False)
newarr = arr[filter_arr]
```

```
print(filter_arr)
```

```
[False, False, True, True]
```

```
print(newarr)
```

```
[43 44]
```

Manipulating arrays

- Filtering: Selecting elements from an existing array using some criteria and creating a new array from them. The operations performed in the last slide can be done in a simpler way:

```
arr = np.array([41, 42, 43, 44])

filter_arr = arr > 42

newarr = arr[filter_arr]
```

```
print(filter_arr)
```

```
[False, False, True, True]
```

```
print(newarr)
```

```
[43 44]
```

Exercise

1. Multiply a 4x3 matrix by a 3x2 matrix (real matrix product), the two matrix must be created with random values;
2. Split the resulting matrix in 1 dimensional arrays;
3. For each array, search for all the values that are greater or equal to 20;
4. Filter each array using the index found in point nr 3;
5. If possible, join the two resulting vectors in a 2d matrix. If it is not possible, insert 0s in order to do it;

Solution

```
A = np.random.normal(loc=3, scale=1.0, size=(4,3))
B = np.random.normal(loc=3, scale=1.0, size=(3,2))
print("A =\n", A)
print("B =\n", A)
```

```
C = A@B
print("C =\n", C)
```

```
D,E = np.array_split(C, 2, 1)
print("D =\n", D)
print("E =\n", E)
```

```
print(D.shape)
D = D[:,0]
E = E[:,0]
print(D)
print(E)
```

```
D_20_index = np.where(D >= 20)
E_20_index = np.where(E >= 20)
print(D_20_index)
print(E_20_index)
```

Solution

```
D_20 = D[D_20_index]
E_20 = E[E_20_index]
print(D_20)
print(E_20)
```

```
if (len(D_20) > len(E_20)):
  m = len(E_20)
  M = len(D_20)
  print(M)
  print(m)
  for i in range(m,M):
      E_20 = np.concatenate((E_20,[0]))
else:
  M = len(E_20)
  m = len(D_20)
  for i in range(m-1,M-1):
      D_20 = np.concatenate((D_20,[0]))

print(D_20.shape)
F_20 = np.stack((D_20, E_20), axis=0)
print(F_20)
```

Pandas

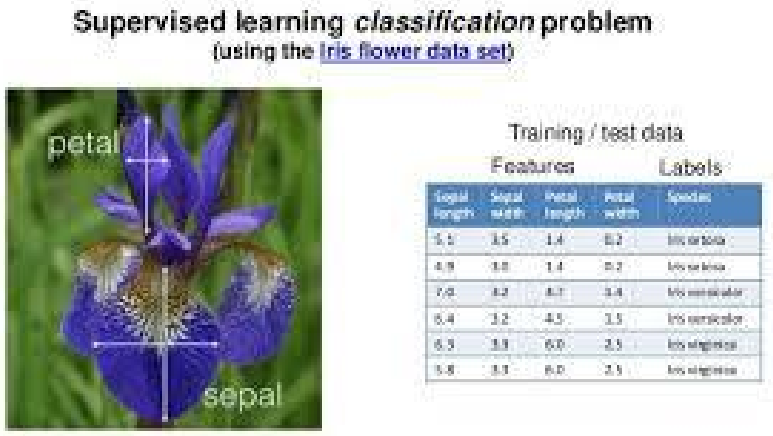
Pandas (“Python Data Analysis”) is a library built upon NumPy array structure. It is used mainly for dataset managing. Its basic object is the **DataFrame** and allows to work with data through tables (i.e. rows observations/samples, columns variables);

Toy Dataset

- We will use the iris dataset, available in the seaborn package;

```
import seaborn as sns
iris = sns.load_dataset("iris")
```

- Dataset containing measurements of different species of iris flowers;



Toy Dataset

- Pandas an also read csv or txt files from an url or local drive e save the data directly in a Dataframe;

```
import pandas as pd
iris_2 =
pd.read_csv("https://raw.githubusercontent.com/mwaskom/seabor
n-data/refs/heads/master/iris.csv")
print(type(iris_2))
```

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

Pandas: checking the data

- The `type()` function let us see what is the object we are investigating;
- The `shape` attribute returns the dimensions of the dataframe, when used on tables it returns the number of rows and columns;
- The `dtypes` attribute returns the variable type of the data contained in the columns;

```
iris.shape
```

```
(150, 5)
```

```
iris.dtypes
```

	0
sepal_length	float64
sepal_width	float64
petal_length	float64
petal_width	float64
species	object

dtype: object

Pandas: checking the data

We can get the columns and rows names using DataFrame attributes;

```
# columns names, returned as an index
iris.columns
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species'],
dtype='object')
```

```
#columns names, returned as an array
iris.columns.values
```

```
array(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species'],
dtype=object)
```

```
# rows names, returned as an index
iris.index
```

```
RangeIndex(start=0, stop=150, step=1)
```

```
# rows names, returned as an array
iris.index.values
```

```
array([  0,   1,   2,   3,   4,   5,…..])
```

Pandas: checking the data

The data contained in the DataFrame can be viewed using its methods.

```
# print first 5 rows
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
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

```
# print first 10 rows
iris.head(10)
```

Pandas: checking the data

You can also view the data by using the columns' name

```
# print the entire column
iris['petal_length']
```

	petal_length
0	1.4
1	1.4
2	1.3
3	1.5
4	1.4
...	...
145	5.2
146	5.0
147	5.2
148	5.4
149	5.1

150 rows × 1 columns

dtype: float64

```
# you can also use the column name as an attribute
Iris.petal_length
```

```
# print the first lines of the selected column
iris['petal_length'].head()
```

Pandas: DataFrame Slicing

We can access part of the dataset by using the [] operator to select rows and columns.

```
# Select specific rows
iris[0:3]
```

	sepal_length	sepal_width	petal_length	petal_width	species
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

```
# Select the first 5 rows
iris[:5]
```

```
# Select the last 6 rows
iris[-6:]
```

```
# Select specific rows
iris[10:13]
```

Pandas: Accessing the data

We can access the data using the following methods:

- **iloc[]** uses indices based on integers;
- **loc[]** uses indices based on labels;

```
# Selection of a columns using iloc
iris.iloc[:, 4]
```

	species
0	setosa
1	setosa
2	setosa
3	setosa
4	setosa
...	...

```
# Slicing rows and columns using iloc
iris.iloc[1:3, 0:2]
```

```
# Slicing rows and columns using loc
# for the rows we use integers, for the columns their name
iris.loc[1:3, ['sepal_length', 'sepal_width']]
```

	sepal_length	sepal_width
1	4.9	3.0
2	4.7	3.2

Pandas: Filtering the data

Data can be filtered using conditions on the values contained in the DataFrame.

```
# Selecting only the rows related to versicolor species
iris[iris['species'] == 'versicolor']
```

	sepal_length	sepal_width	petal_length	petal_width	species
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
55	5.7	2.8	4.5	1.3	versicolor

```
# Selecting only the rows not related to versicolor species
iris[iris['species'] != 'versicolor’]
```

	sepal_length	sepal_width	petal_length	petal_width	species
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

Pandas: Filtering the data

Data can be filtered using conditions on the values contained in the DataFrame.
Multiple conditions can be used.

```
# Select all the samples of setosa species having a petal length greater than
# 1.5 cm
iris[(iris['species'] == 'setosa') & (iris['petal_length'] > 1.5)]
```

	sepal_length	sepal_width	petal_length	petal_width	species
5	5.4	3.9	1.7	0.4	setosa
11	4.8	3.4	1.6	0.2	setosa
18	5.7	3.8	1.7	0.3	setosa
20	5.4	3.4	1.7	0.2	setosa
23	5.1	3.3	1.7	0.5	setosa

Exercise

Select all the samples of virginica samples having sepal_width less or equal to 3.0 and print only the columns sepal_length, sepal_width, species.

Exercise

Select all the samples of virginica samples having sepal_width less or equal to 3.0 and print only the columns sepal_length, sepal_width, species.

Solution

```
iris[(iris.species == 'virginica') & (iris.sepal_width <= 3.0)][['sepal_length', 'sepal_width', 'species']]
```

Pandas: statistics on the data

Pandas' dataframes offer different attributes, methods and functions to obtain statistics on the data, useful to prepar them for the analysis

- Unique values in a column

```
pd.unique(iris['species'])
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

```
# the values can be saved in an array
species = pd.unique(iris['species'])
print(species)
```

```
['setosa' 'versicolor' 'virginica']
```

- Number of unique values in a column

```
# We can use the array created previously
len(species)
```

```
3
```

```
# We can use the nunique() method
iris['species'].nunique()
```

```
3
```

- Descriptive statistics

Different values such as mean and median on the data contained in columns of numeric type.

```
iris['sepal_length'].describe()
```

	sepal_length
count	150.000000
mean	5.843333
std	0.828066
min	4.300000
25%	5.100000
50%	5.800000
75%	6.400000
max	7.900000

dtype: float64

```
# We can print specific values
iris['sepal_length'].median()
```

```
5.8
```

- Grouping data

Data can be grouped by values contained in the columns

```
iris.groupby('species')
```

The resulting dataframe can be saved in another one

```
species_groupby = iris.groupby('species')
species_groupby.head(2)
```

The operations are performed on each group!

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
50	7.0	3.2	4.7	1.4	versicolor
51	6.4	3.2	4.5	1.5	versicolor
100	6.3	3.3	6.0	2.5	virginica
101	5.8	2.7	5.1	1.9	virginica

- Grouping data- Statistics

Once grouped, we can count the total samples for each values of the column.

```
species_groupby.count()
```

species	sepal_length	sepal_width	petal_length	petal_width
setosa	50	50	50	50
versicolor	50	50	50	50
virginica	50	50	50	50

In a similar way we can obtain the descriptive statistics:

```
species_groupby.describe()
```

	sepal_length								sepal_width		...	petal_length		petal_width						
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std	min	25%	50%	75%	max
species																				
setosa	50.0	5.006	0.352490	4.3	4.800	5.0	5.2	5.8	50.0	3.428	...	1.575	1.9	50.0	0.246	0.105386	0.1	0.2	0.2	0.3	0.6
versicolor	50.0	5.936	0.516171	4.9	5.600	5.9	6.3	7.0	50.0	2.770	...	4.600	5.1	50.0	1.326	0.197753	1.0	1.2	1.3	1.5	1.8
virginica	50.0	6.588	0.635880	4.9	6.225	6.5	6.9	7.9	50.0	2.974	...	5.875	6.9	50.0	2.026	0.274650	1.4	1.8	2.0	2.3	2.5

3 rows × 32 columns

We can group by multiple columns

```
iris.groupby(['species', 'sepal_length'])['petal_length'].count()
```

		petal_length
species	sepal_length	
setosa	4.3	1
	4.4	3
	4.5	1
	4.6	4
	4.7	2
	4.8	5
	4.9	4

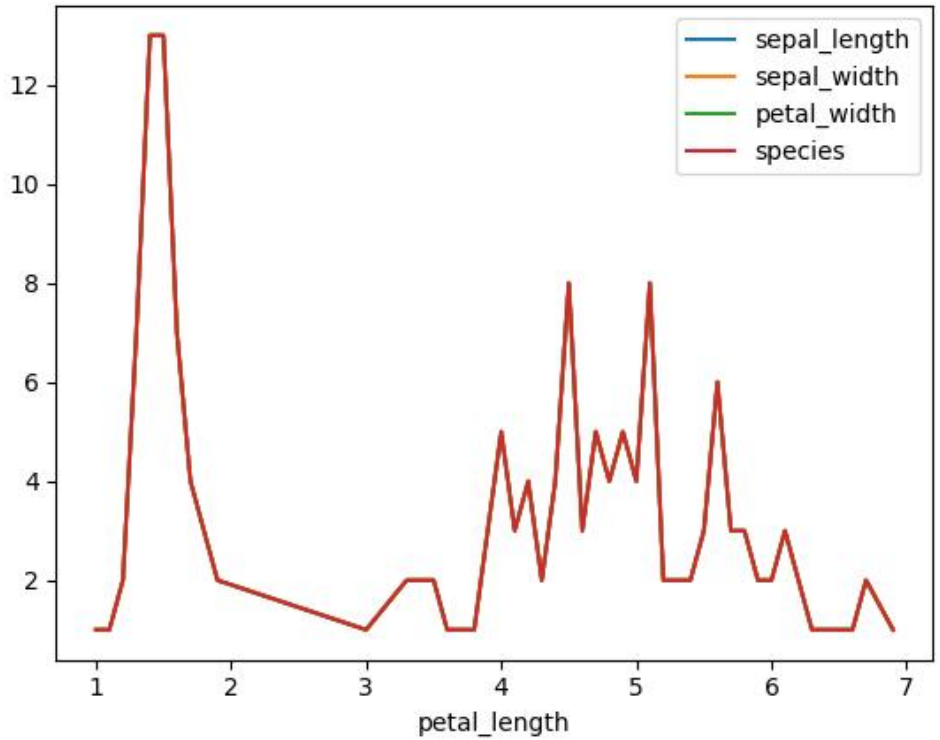
We can also obtain how many samples are in a specific group, separated bu the values of a column.

```
species_seplen_count = iris.groupby(['species',
'sepal_length'])['petal_length'].count().unstack('sepal_length')
print(species_seplen_count)
print(type(species_seplen_count))
```

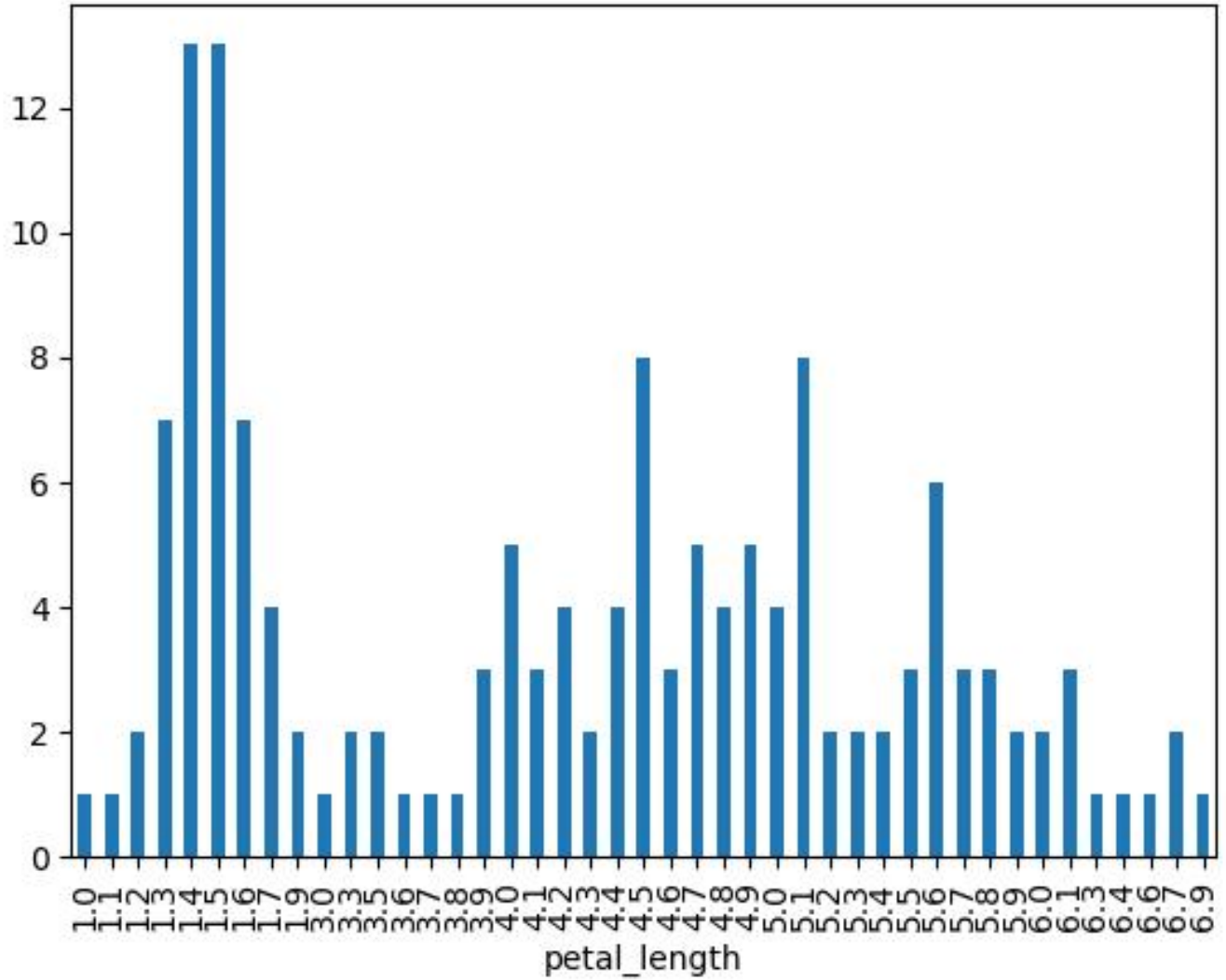
Pandas: plotting the data

Pandas is able to quickly plot the data

```
petlen_count = iris.groupby('petal_length')['species'].count()
petlen_count.plot()
```

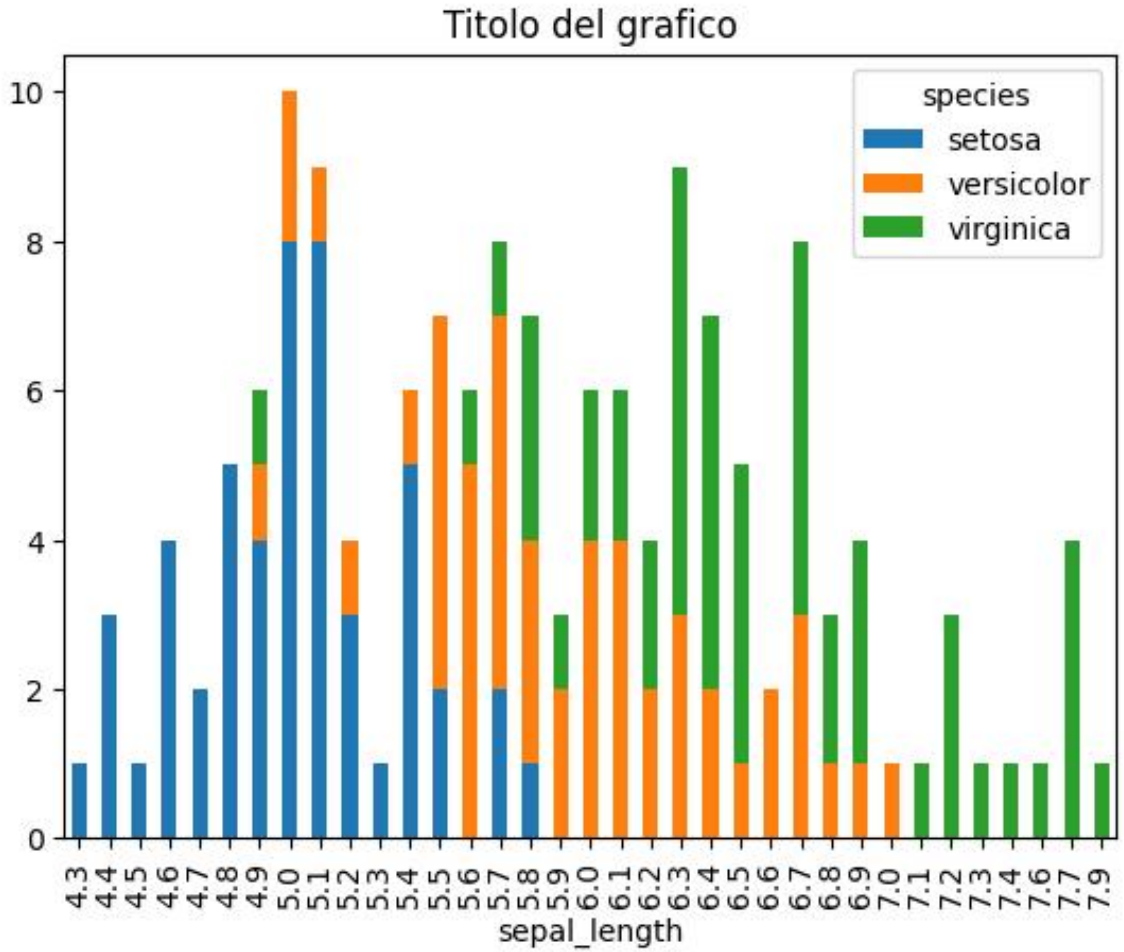


```
petlen_count.plot(kind = 'bar')
```



```
species_seplen_count = iris.groupby(['species',
'sepal_length'])['petal_length'].count().unstack('species')

species_seplen_count.plot(kind = 'bar', stacked=True,
title = 'Titolo del grafico')
```



Exercise

Task 1: Make a bar plot of the mean sepal sizes for each species
Task 2: Find the min, max and mean of petal sizes for each species

Solution

Task 1: Make a bar plot of the mean sepal sizes for each species

```
# Group by species
iris_species = iris.groupby("species")
# get descriptive statistics on it
stats = iris_species.describe()
```

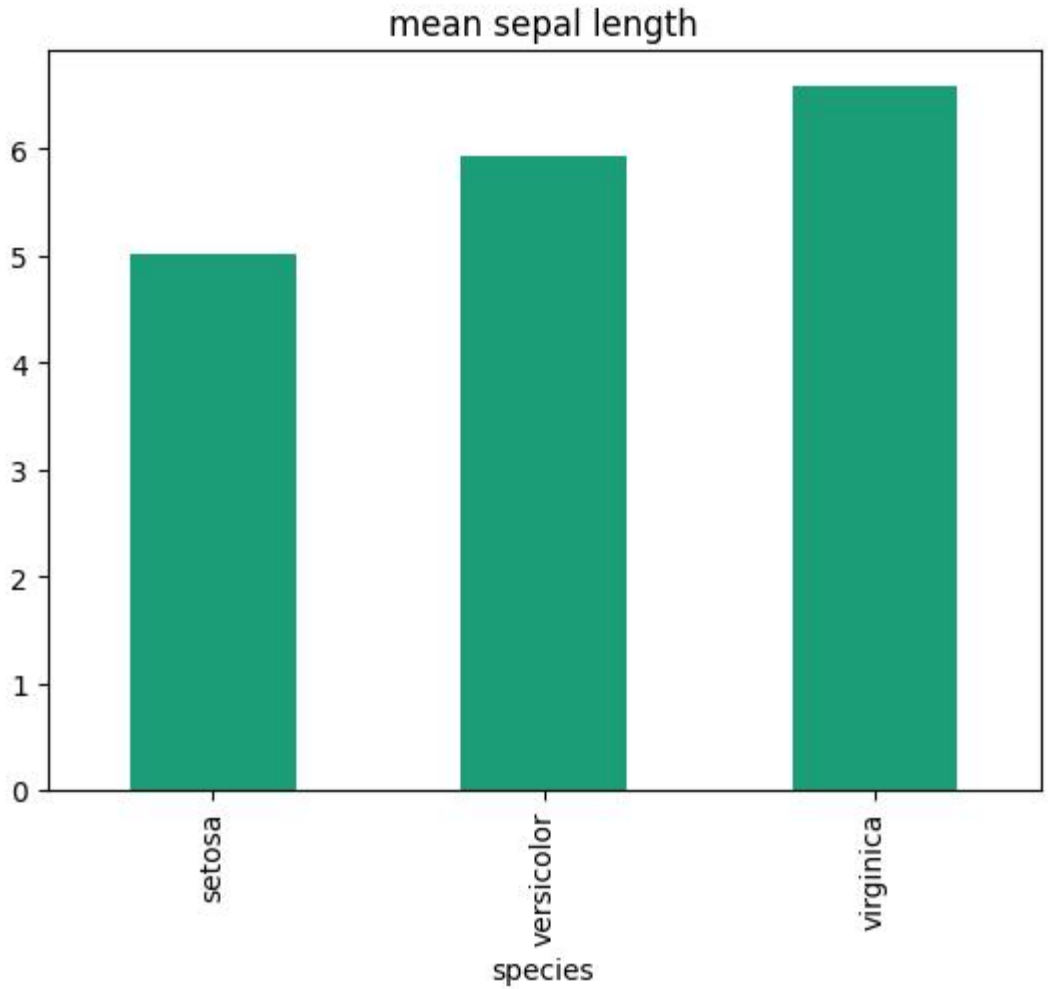
```
# save mean sepal sizes
mean_sep_width = stats.sepal_width["mean"]
mean_sep_length = stats.sepal_length["mean"]
```

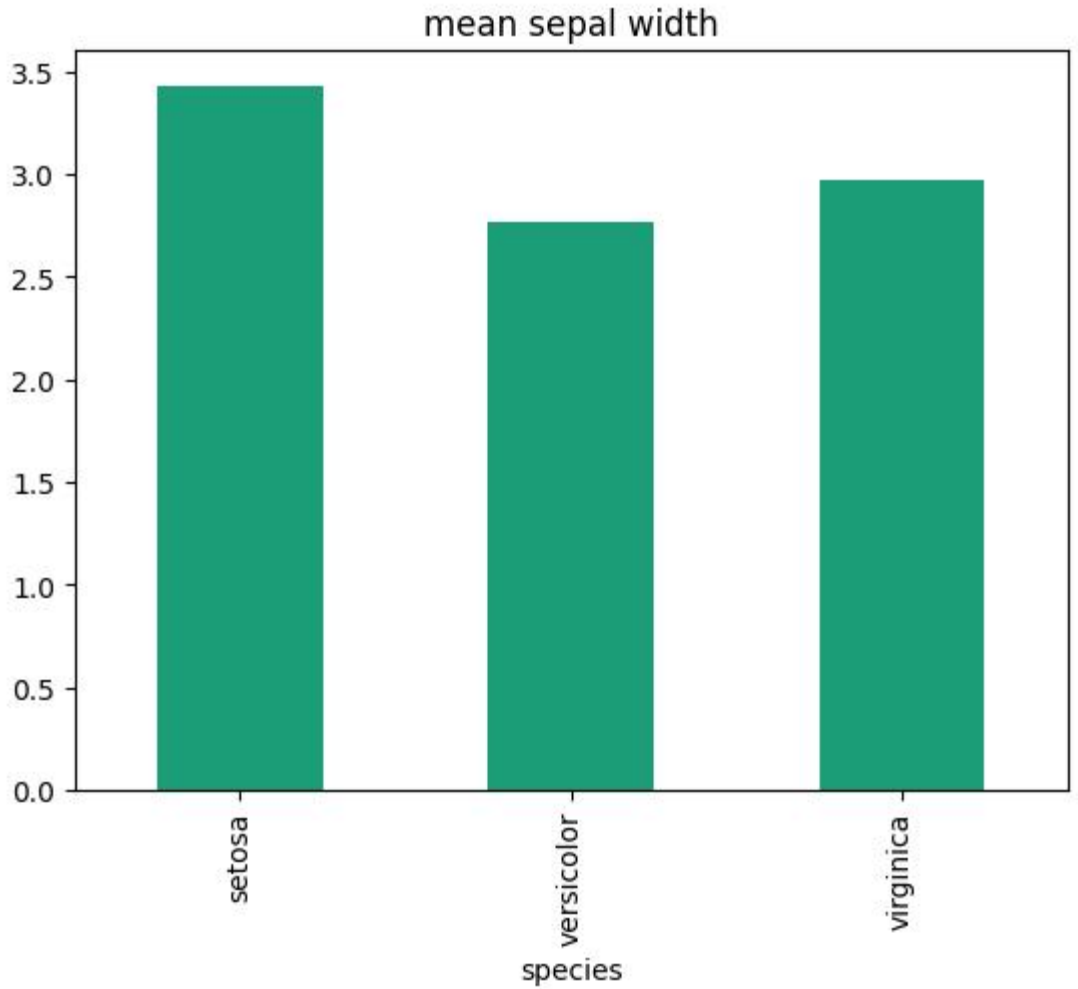
```
# print barplots (separate)
mean_sep_length.plot(kind='bar', colormap='Dark2', title='mean sepal
length')
```

```
mean_sep_width.plot(kind='bar', colormap='Dark2', title='mean sepal
width' )
```

Solution

Task 1: Make a bar plot of the mean sepal sizes for each species

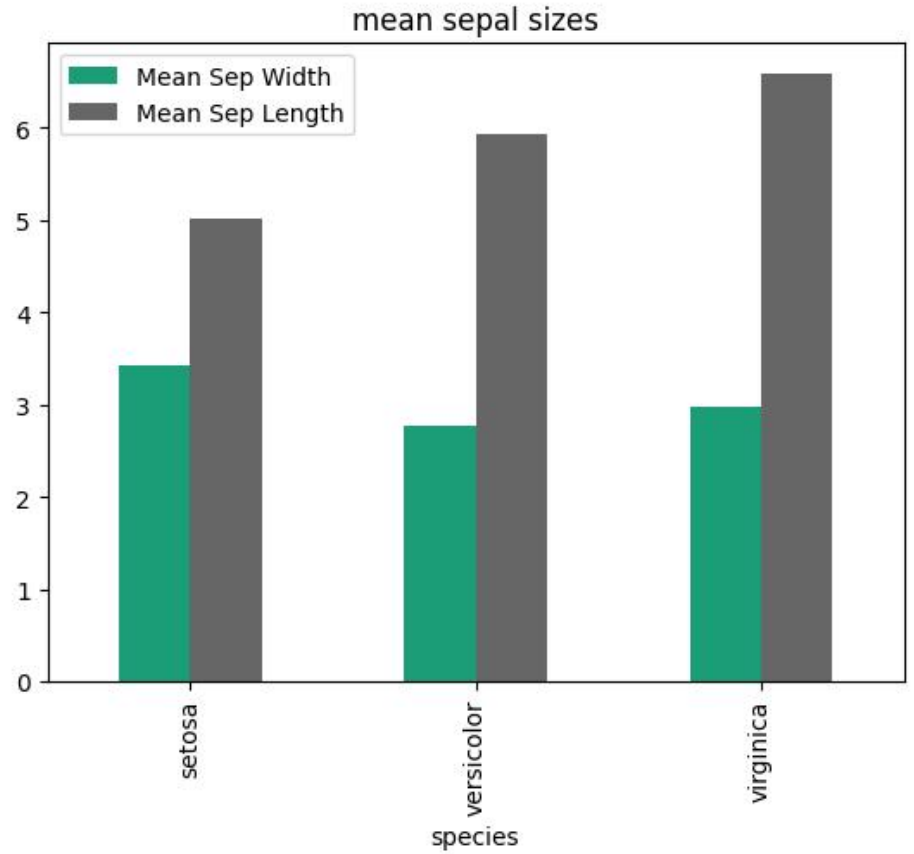




Solution

Task 1: Make a bar plot of the mean sepal sizes for each species

```
# how to do it in a single plot
sepal_sizes = pd.merge(left=mean_sep_width, right=mean_sep_length, on='species')
sepal_sizes.columns = ['Mean Sep Width', 'Mean Sep Length']
sepal_sizes.plot(kind='bar', colormap='Dark2', title='mean sepal sizes',  )
```



Solution

Task 2: Find the min, max and mean of petal sizes for each species

```
# get all the petal sizes for all species
petal_sizes = stats[['petal_length','petal_width']]
print(petal_sizes)
```

```
# we are interested only in min max and mean
print(petal_sizes.petal_length[["min","max","mean"]])
print(petal_sizes.petal_width[["min","max","mean"]])
```

```
              min  max   mean
species
setosa        1.0  1.9  1.462
versicolor    3.0  5.1  4.260
virginica     4.5  6.9  5.552
              min  max   mean
species
setosa        0.1  0.6  0.246
versicolor    1.0  1.8  1.326
virginica     1.4  2.5  2.026
```

THANKS!

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani