Projet Analyse De Donnée

L3

Hossein Khani

Content:

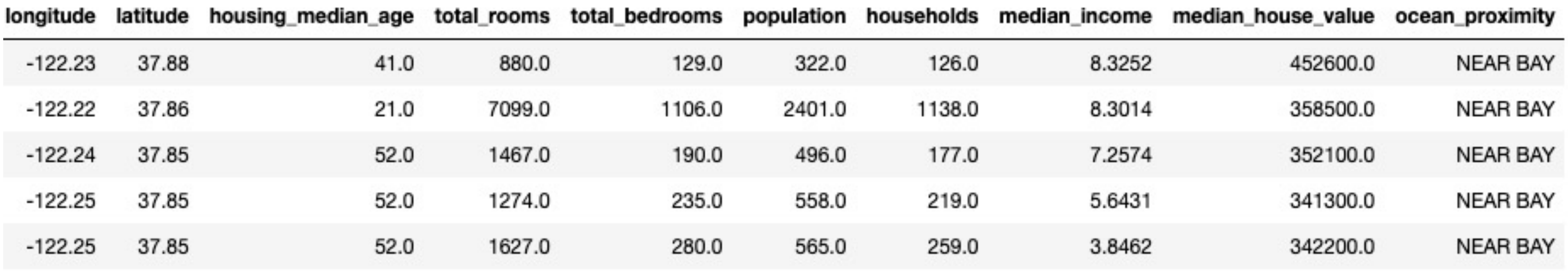
- ➢ Why Data Analyses
- ➢ Data Manipulation (Pandas Library)
- ➢ Data Visualisation (Matplotlib,Pyplot, Seaborn)
- ➢ Linear Regression
- ➢ Principle Component Analysis
- ➢ Non-Negative Matrix Factorization
- ➢ Orthogonal Matching pursuit

NEEDS:

- ➢ Basic Python Skills (Lists, Dictionaries, Functions, methods,….)
- ➢ Working with DataFrames (Data Cleaning and manipulation with Pandas Library)
- ➢ Working with Matrices (Numpy Library)
- ➢ Mathematics behind Machine learning Techniques (Mostly probability and statistics)
- ➢ Machine learning library (Scipy or Sklearn)

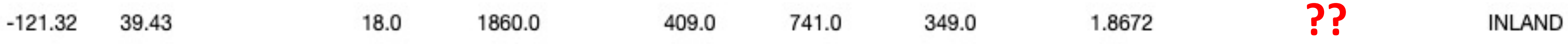
QUESTION:

Input:



longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

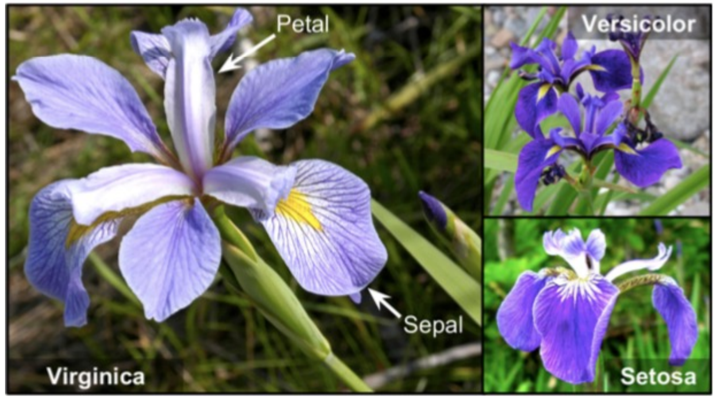
Output:



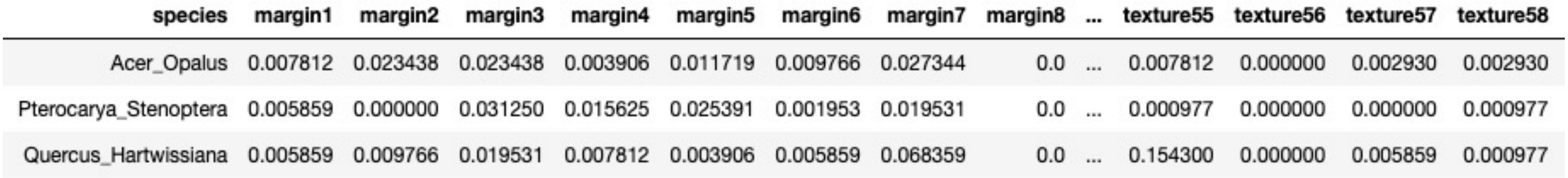
-121.32	39.43	18.0	1860.0	409.0	741.0	349.0	1.8672	??	INLAND

REGRESSION

QUESTION:



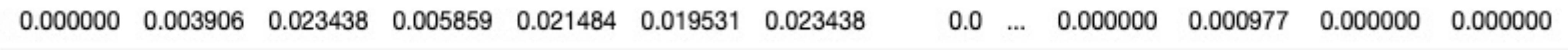
Input:



species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	...	texture55	texture56	texture57	texture58
Acer_Opalus	0.007812	0.023438	0.023438	0.003906	0.011719	0.009766	0.027344	0.0	...	0.007812	0.000000	0.002930	0.002930
Pterocarya_Stenoptera	0.005859	0.000000	0.031250	0.015625	0.025391	0.001953	0.019531	0.0	...	0.000977	0.000000	0.000000	0.000977
Quercus_Hartwissiana	0.005859	0.009766	0.019531	0.007812	0.003906	0.005859	0.068359	0.0	...	0.154300	0.000000	0.005859	0.000977

Output:

??



0.000000	0.003906	0.023438	0.005859	0.021484	0.019531	0.023438	0.0	...	0.000000	0.000977	0.000000	0.000000

CLASSIFICATION

Install Python

- https://www.python.org/
- pip Python package management system : **python3 -m pip --version**
- install jupyter notebook: **python3 -m pip install -U jupyter**
- Install pandas: **pip install pandas**

- The Jupyter Notebook is the original web application for creating and sharing computational documents.
- Pandas the main tool of data analyse
- Pandas permits us to import data from various sources for example (CSV), and manipute them.

DataFrames:

- **https://insights.stackoverflow.com/survey**

How to use Pandas to work with DataFrame……

1. How to read data from csv file,
2. Take a look at the datafram,
3. Where dataframe comes from, its equivalent in python
4. Series objects and accessing multi-columns
5. Indexing
6. Accessing rows in DataFrames
7. Setting index for data frame
8. Changing columns’ names
9. Changing single row’s values

Numpy:

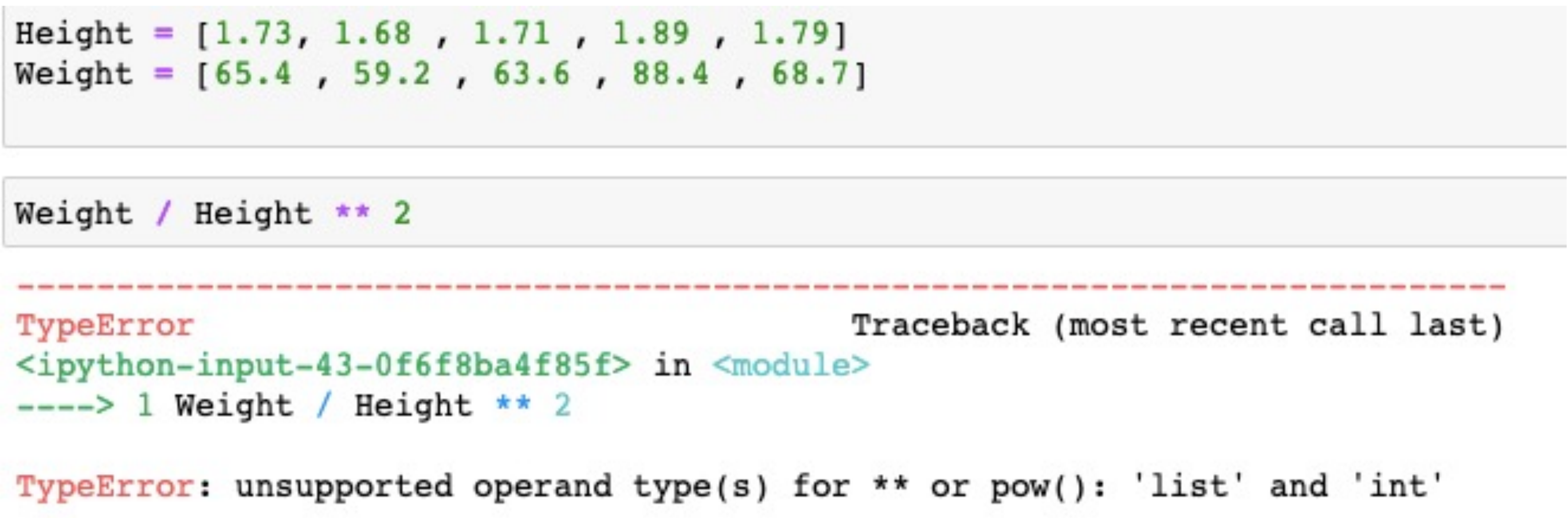
As a Data Analyst how to collect data?

- List?
- Collection of values
- Hold different types
- Change, add, remove

- What we need more?
- Mathematical operations over collections
- Speed

Numpy:

Body mass Index:



```
Height = [1.73, 1.68 , 1.71 , 1.89 , 1.79]
Weight = [65.4 , 59.2 , 63.6 , 88.4 , 68.7]
```

```
Weight / Height ** 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-0f6f8ba4f85f> in <module>
----> 1 Weight / Height ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

To Solve: Looping over elements? → Not fast and efficient

Numpy (numeric python):

Solution?

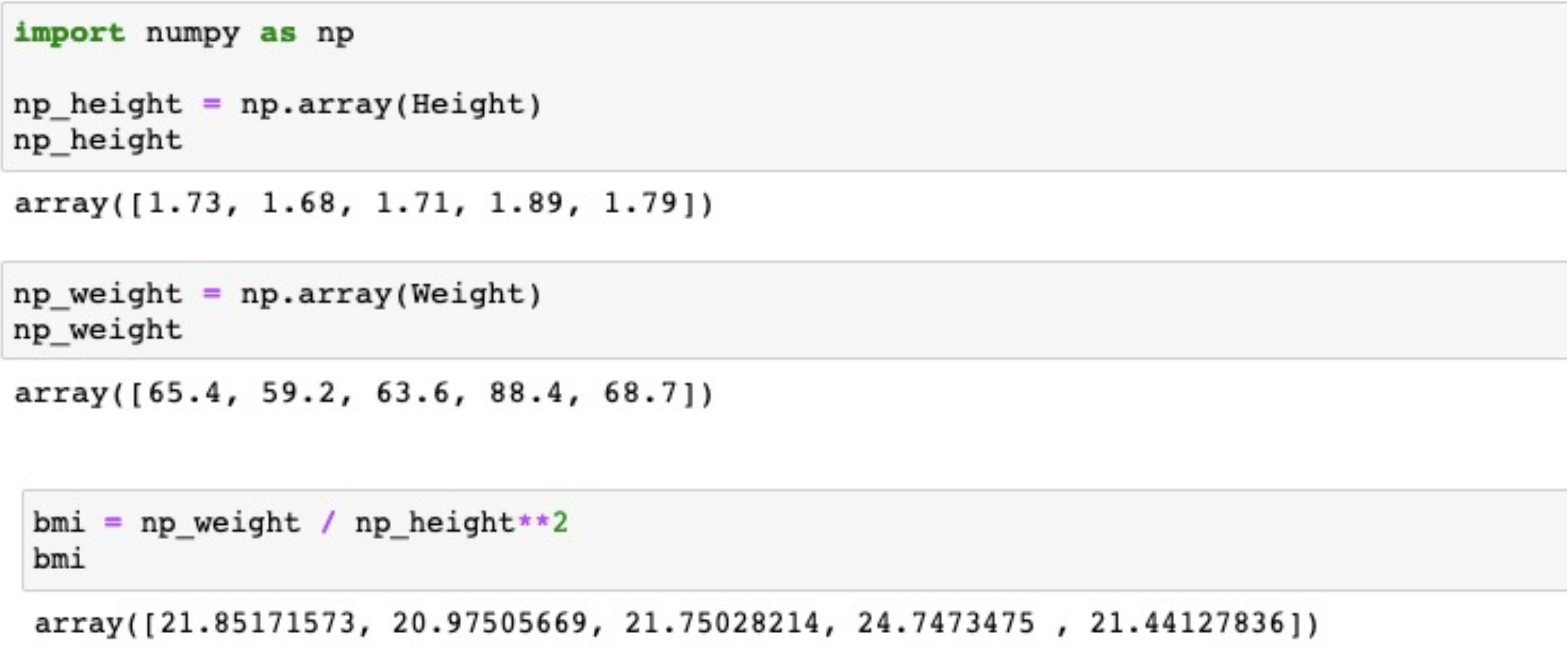
nympy arrays:

- Alternative to python lists
- Calculations over entire arrays
- Easy and Fast

To Install:

```
pip3 install numpy
```

Numpy



```
import numpy as np

np_height = np.array(Height)
np_height
```

```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
np_weight = np.array(Weight)
np_weight
```

```
array([65.4, 59.2, 63.6, 88.4, 68.7])
```

```
bmi = np_weight / np_height**2
bmi
```

```
array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])
```

Numpy

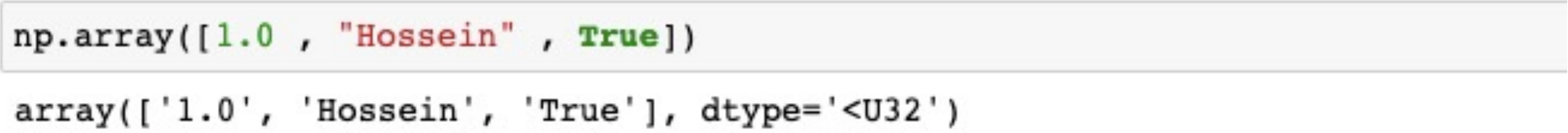
Python is able to treat numpy arrays as single elements.

Where the speed comes from?

Numpy arrays collect values of the same type:

- Either integer
- Either float
- String
- …..

Numpy (Remarks)



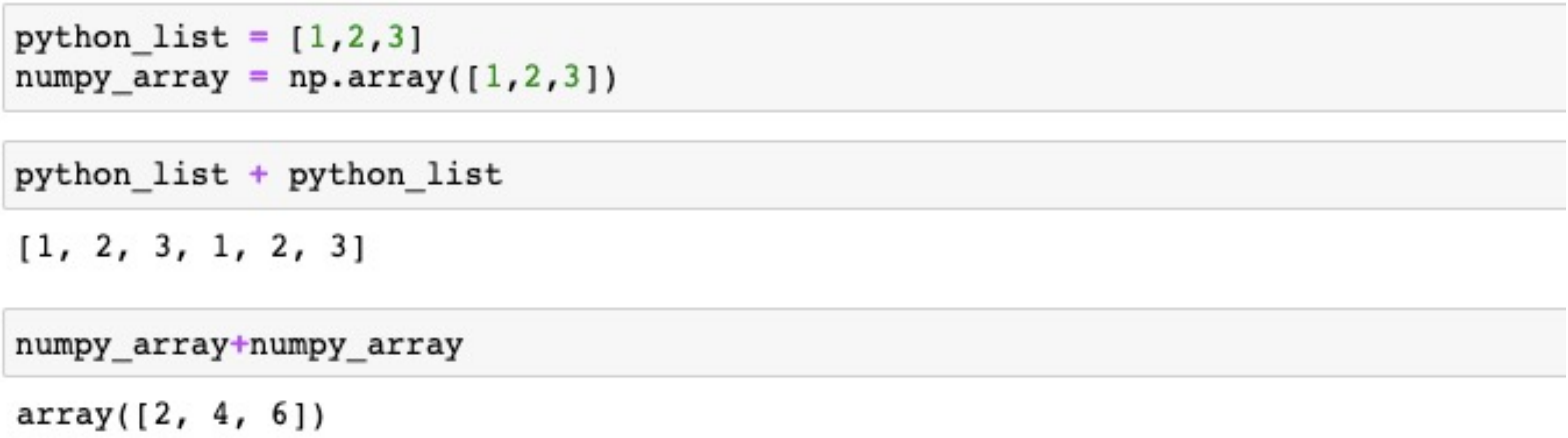
```
np.array([1.0 , "Hossein" , True])
```

```
array(['1.0', 'Hossein', 'True'], dtype='<U32')
```

- Nympy array, is a data type in python.
- It has its own methods.
- These methods might act differently on arrays compared to other types.

Numpy (Remarks)

Example:



```
python_list = [1,2,3]
numpy_array = np.array([1,2,3])
```

```
python_list + python_list
```

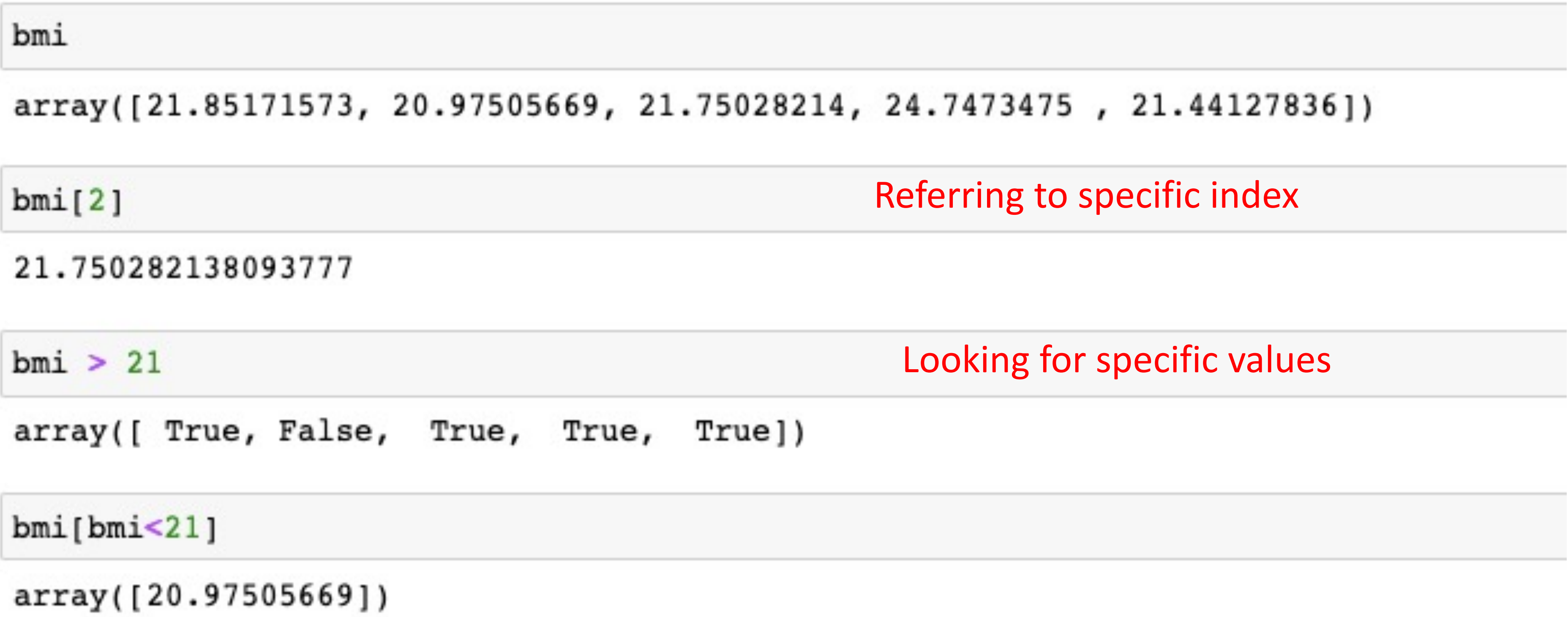
```
[1, 2, 3, 1, 2, 3]
```

```
numpy_array+numpy_array
```

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

Numpy (Subsetting)

Example:

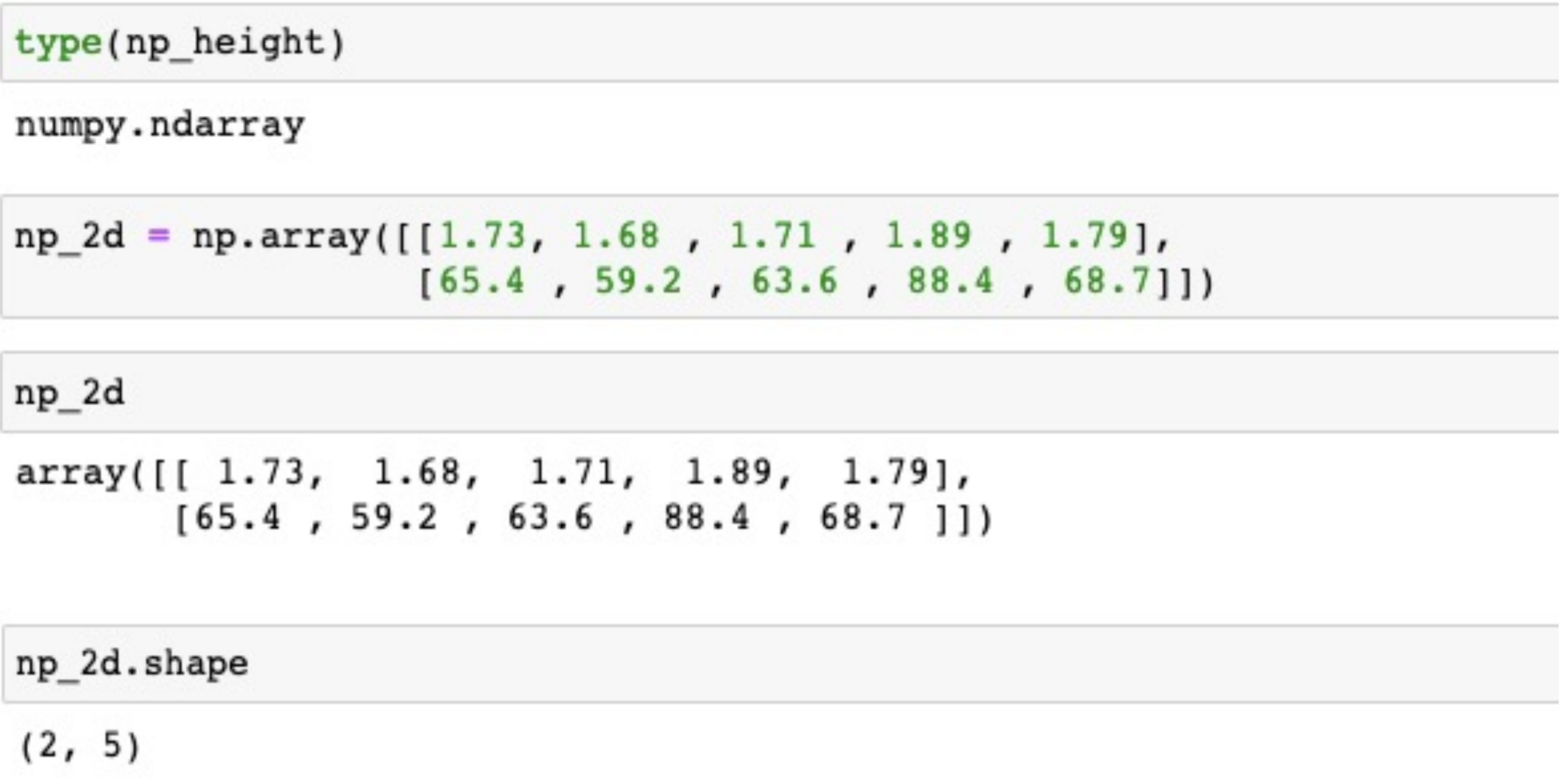


```
bmi

array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])

bmi[2]

21.750282138093777

bmi > 21

array([ True, False,  True,  True,  True])

bmi[bmi<21]

array([20.97505669])
```

Referring to specific index

Looking for specific values

Numpy (2D)



```
type(np_height)
```

```
numpy.ndarray
```

```
np_2d = np.array([[1.73, 1.68 , 1.71 , 1.89 , 1.79],
                  [65.4 , 59.2 , 63.6 , 88.4 , 68.7]])
```

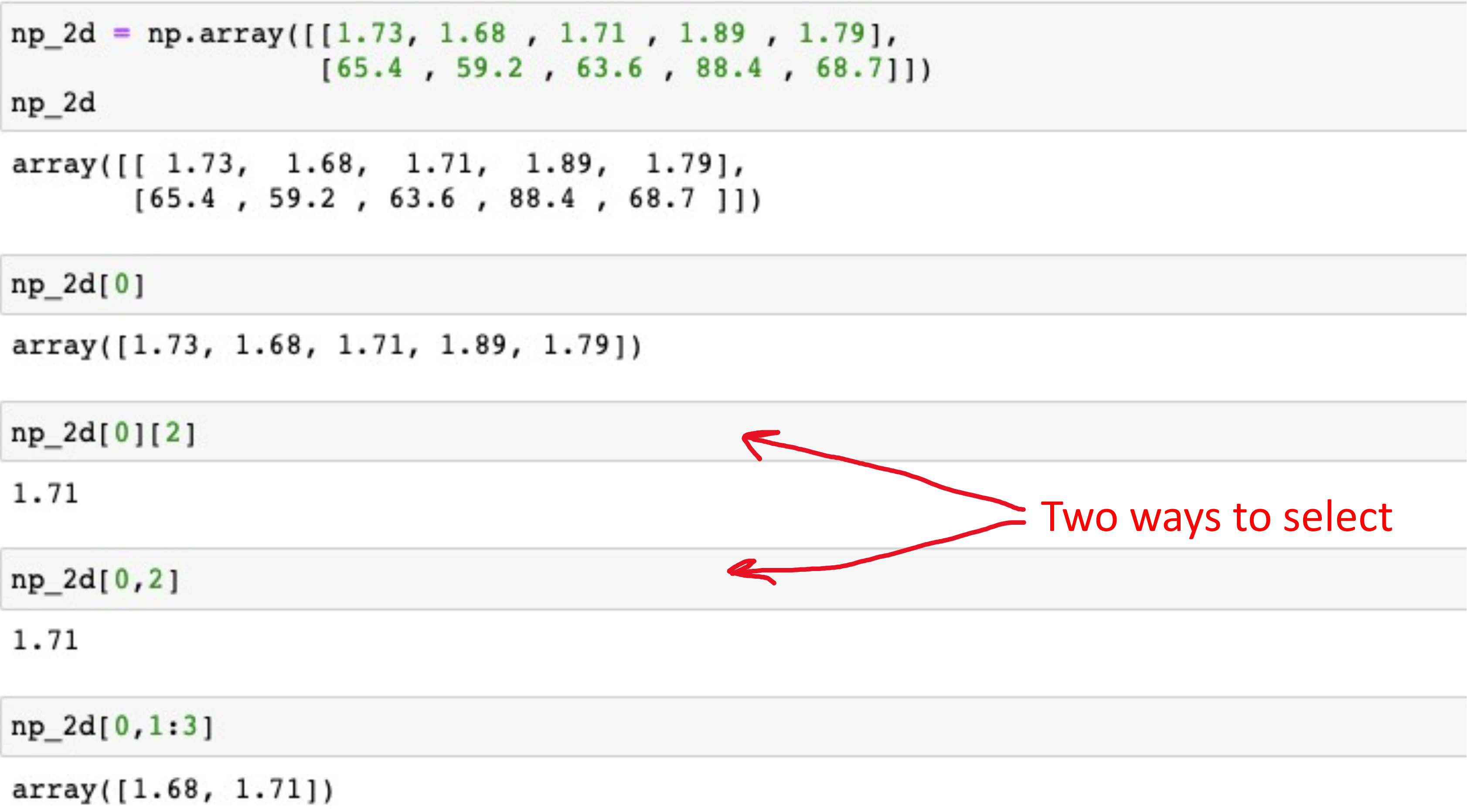
```
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np_2d.shape
```

```
(2, 5)
```

Numpy (2D)



```
np_2d = np.array([[1.73, 1.68 , 1.71 , 1.89 , 1.79],
                  [65.4 , 59.2 , 63.6 , 88.4 , 68.7]])
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np_2d[0]
```

```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
np_2d[0][2]
```

```
1.71
```

```
np_2d[0,2]
```

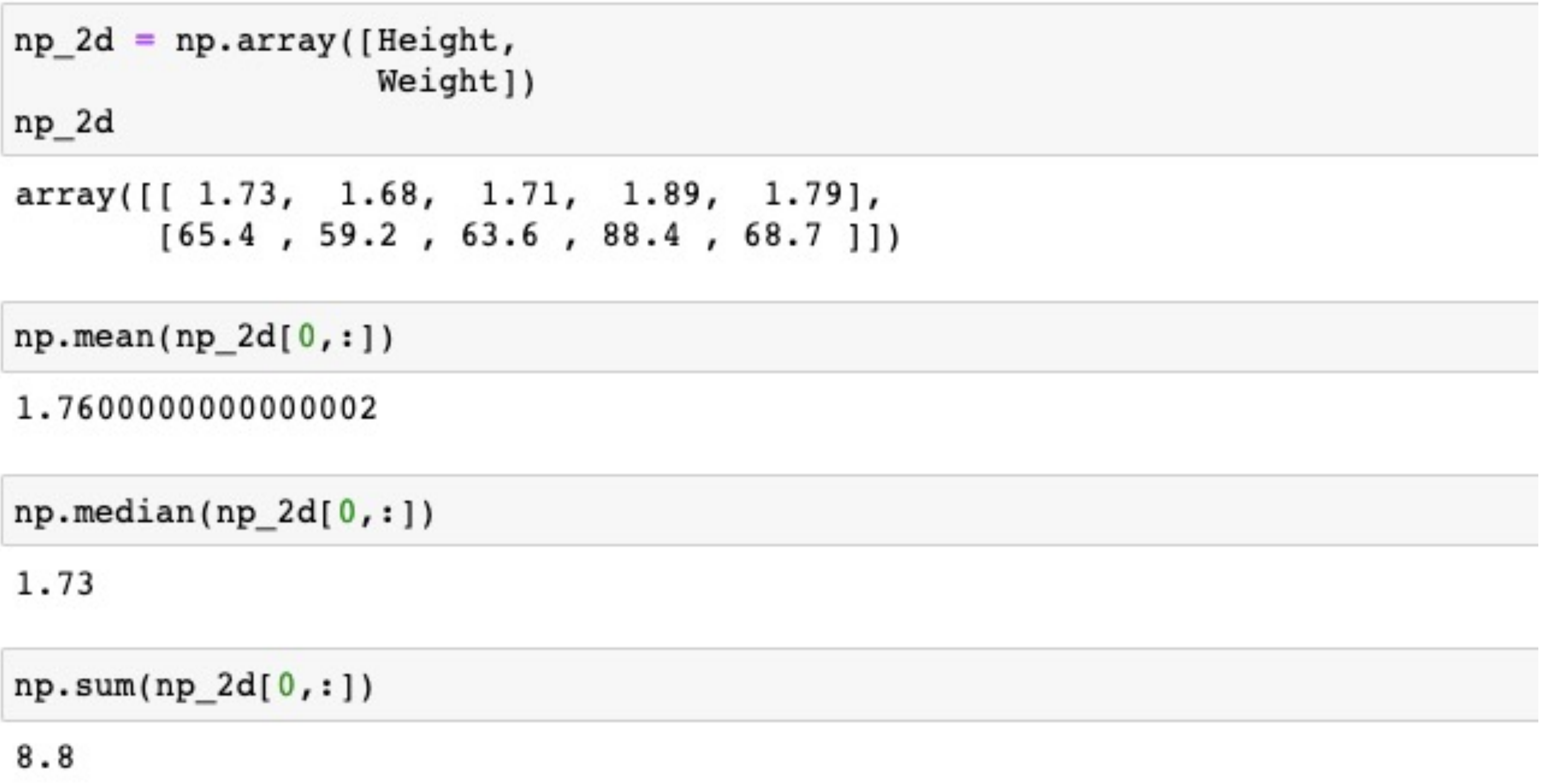
```
1.71
```

```
np_2d[0,1:3]
```

```
array([1.68, 1.71])
```

Two ways to select

Numpy (Basic Statistics)



```
np_2d = np.array([Height,
                  Weight])
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np.mean(np_2d[0,:])
```

```
1.7600000000000002
```

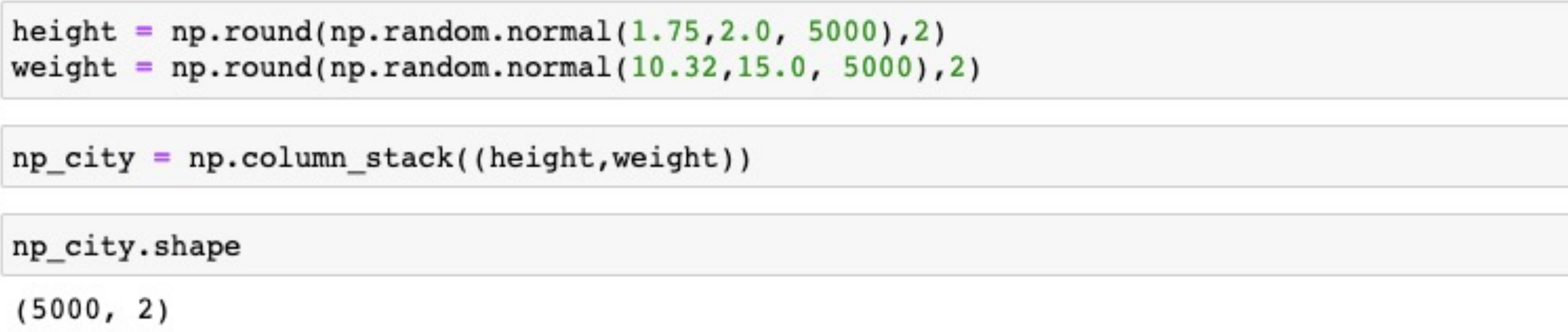
```
np.median(np_2d[0,:])
```

```
1.73
```

```
np.sum(np_2d[0,:])
```

```
8.8
```

Numpy (Data Generation)



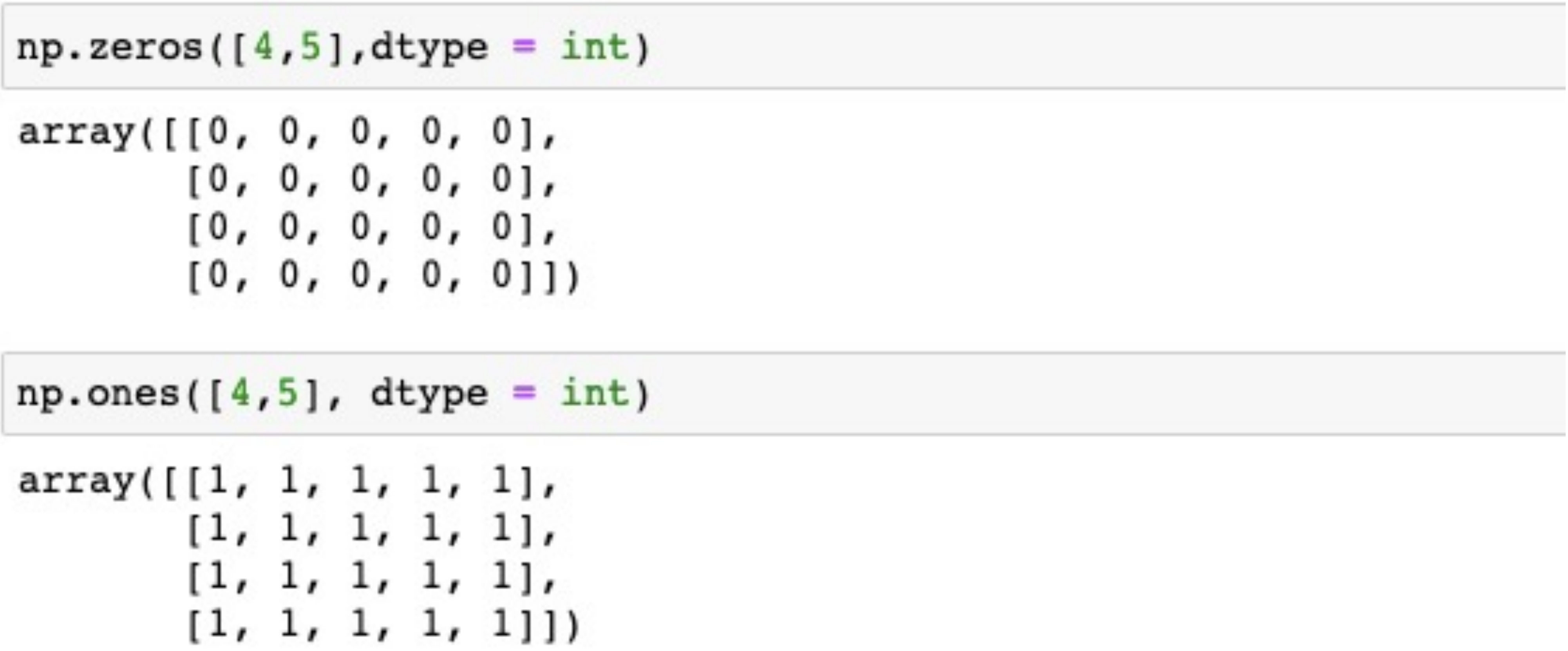
```
height = np.round(np.random.normal(1.75,2.0, 5000),2)
weight = np.round(np.random.normal(10.32,15.0, 5000),2)
```

```
np_city = np.column_stack((height,weight))
```

```
np_city.shape
```

```
(5000, 2)
```

Numpy (Data Generation)

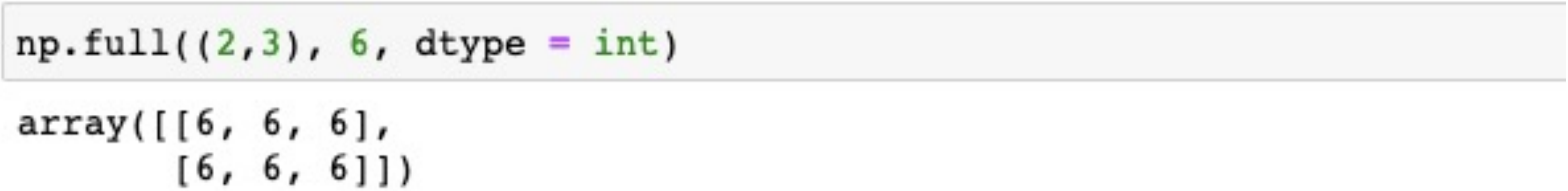


```
np.zeros([4,5],dtype = int)
```

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

```
np.ones([4,5], dtype = int)
```

```
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```



```
np.full((2,3), 6, dtype = int)
```

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

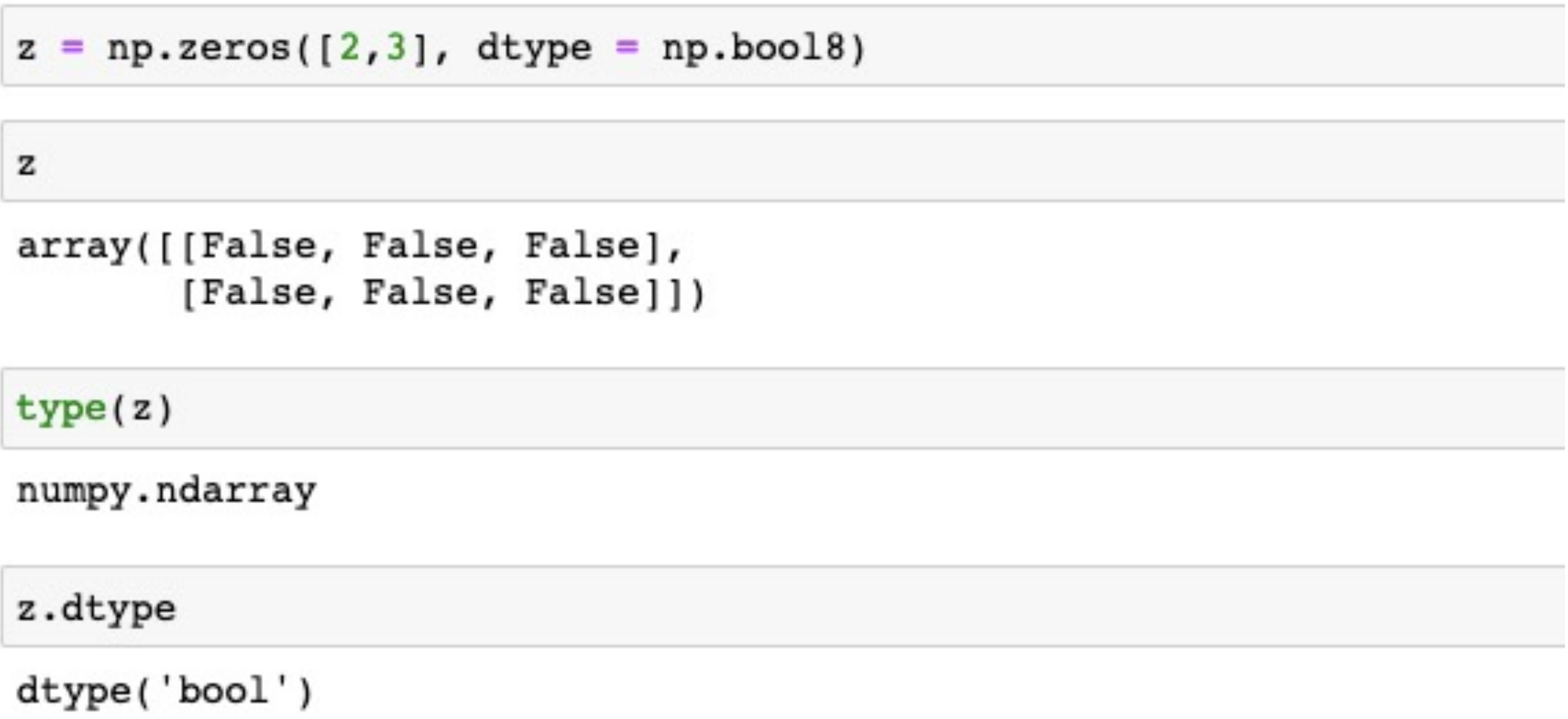
Numpy (Dtype)

- Python types: int, float, bool,…

  Their size depends on the platform they are applied to…

- Dtypes: numpy numerical types are instances of dtype objects. The numpy types have fixed-sizes.

  np.int32, np.int64, np.bool8, np.float32, np.float64



```
z = np.zeros([2,3], dtype = np.bool8)
```

```
z
```

```
array([[False, False, False],
       [False, False, False]])
```

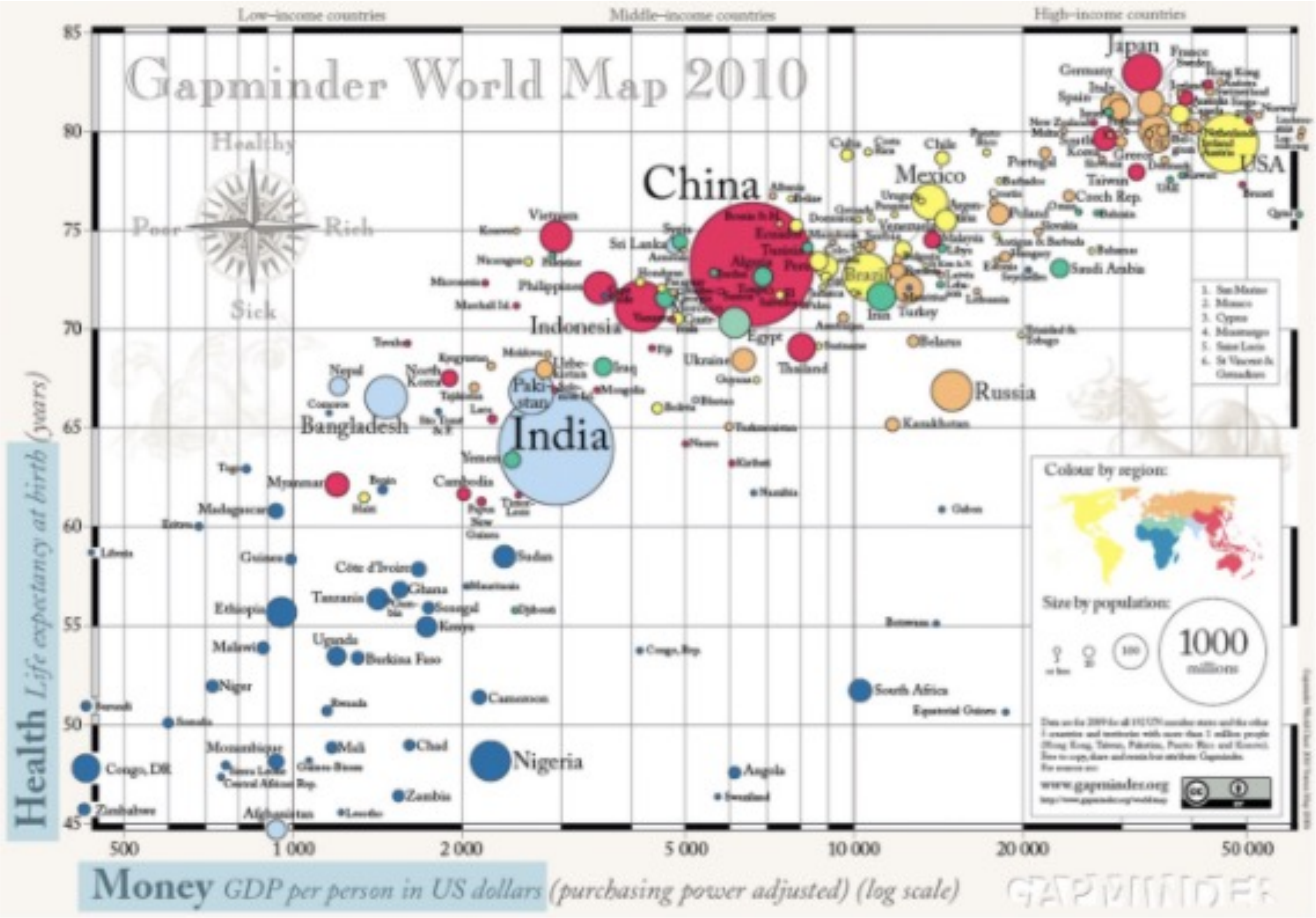
```
type(z)
```

```
numpy.ndarray
```

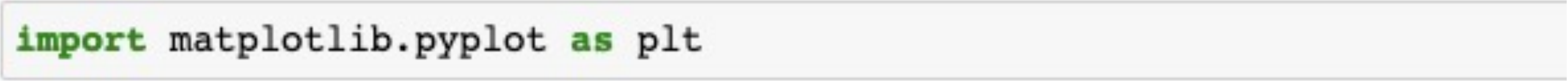
```
z.dtype
```

```
dtype('bool')
```

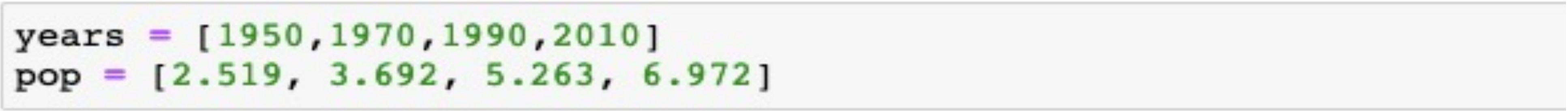
Data Visualisation:



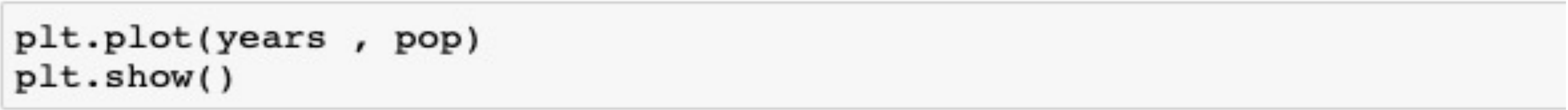
The most important visualization library : Matplotlib:



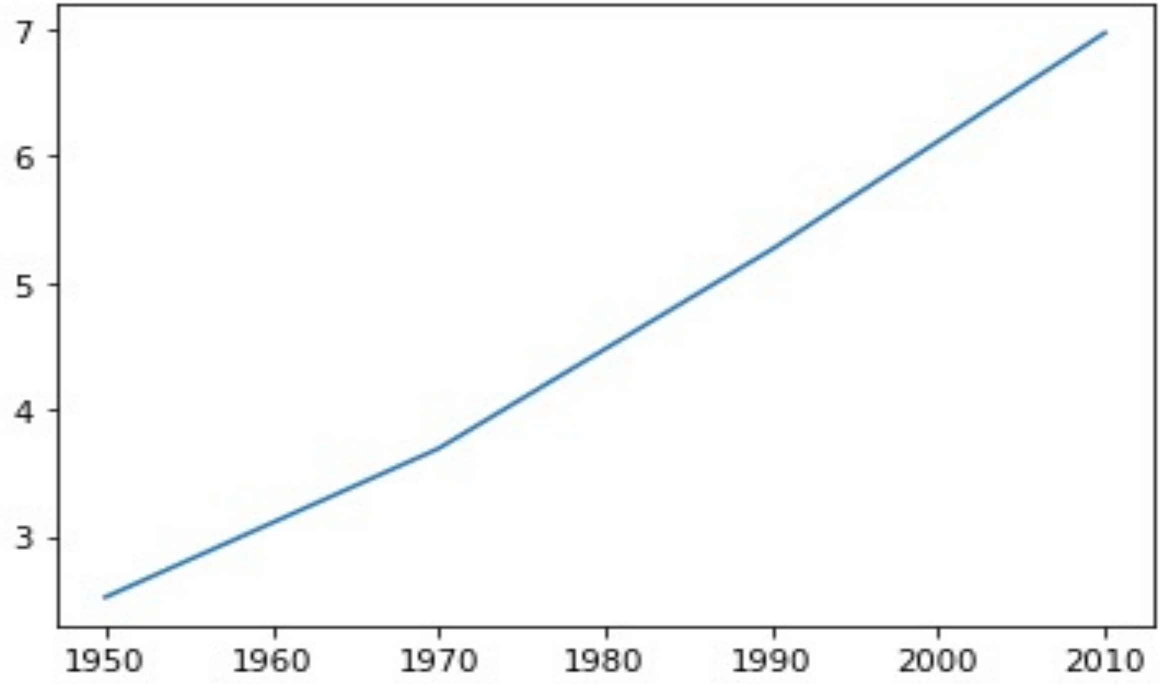
```
import matplotlib.pyplot as plt
```



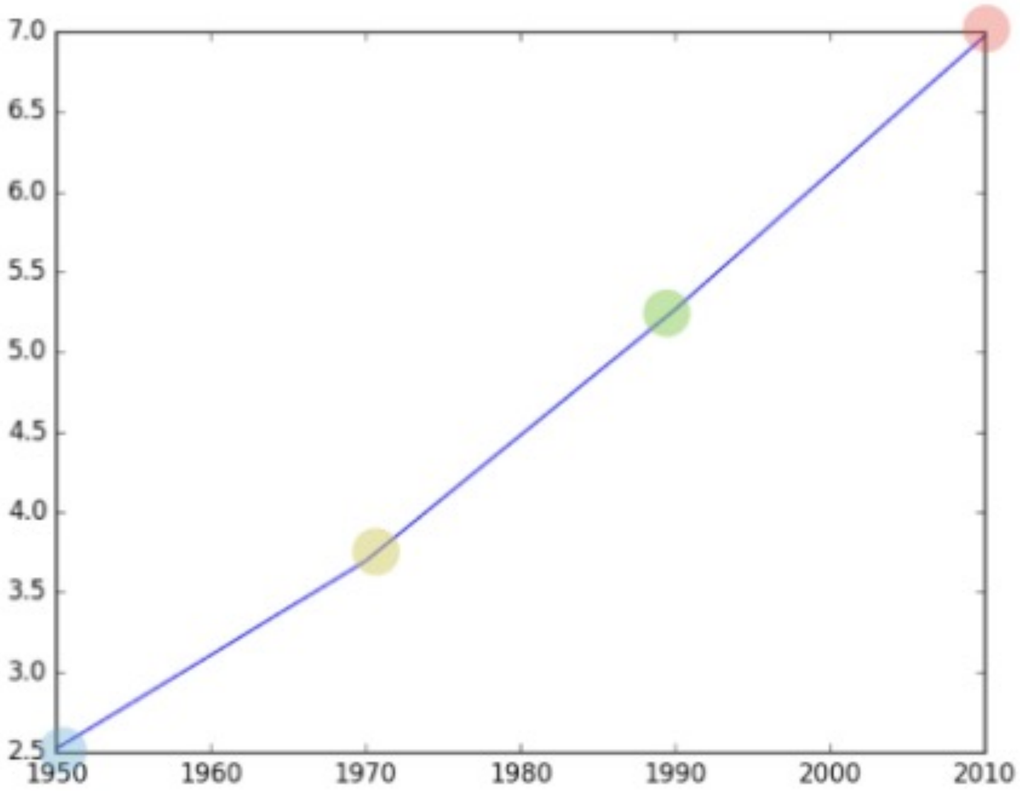
```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

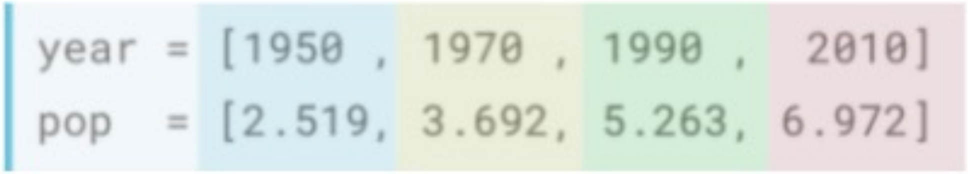


```
plt.plot(years , pop)
plt.show()
```



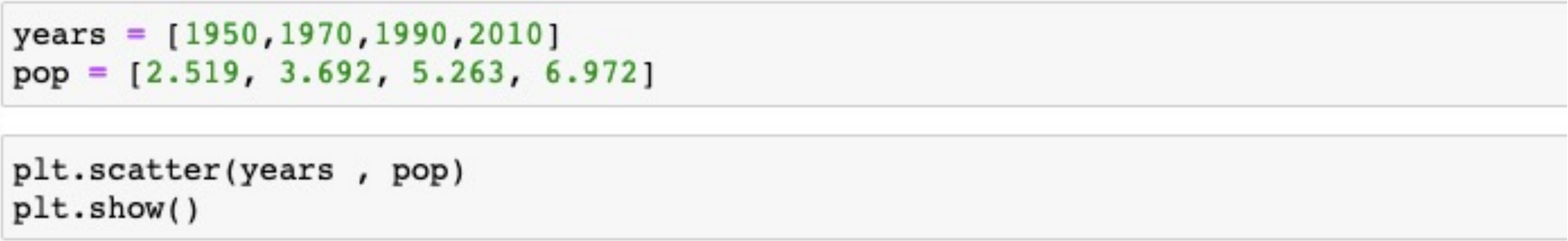
plt.plot() :





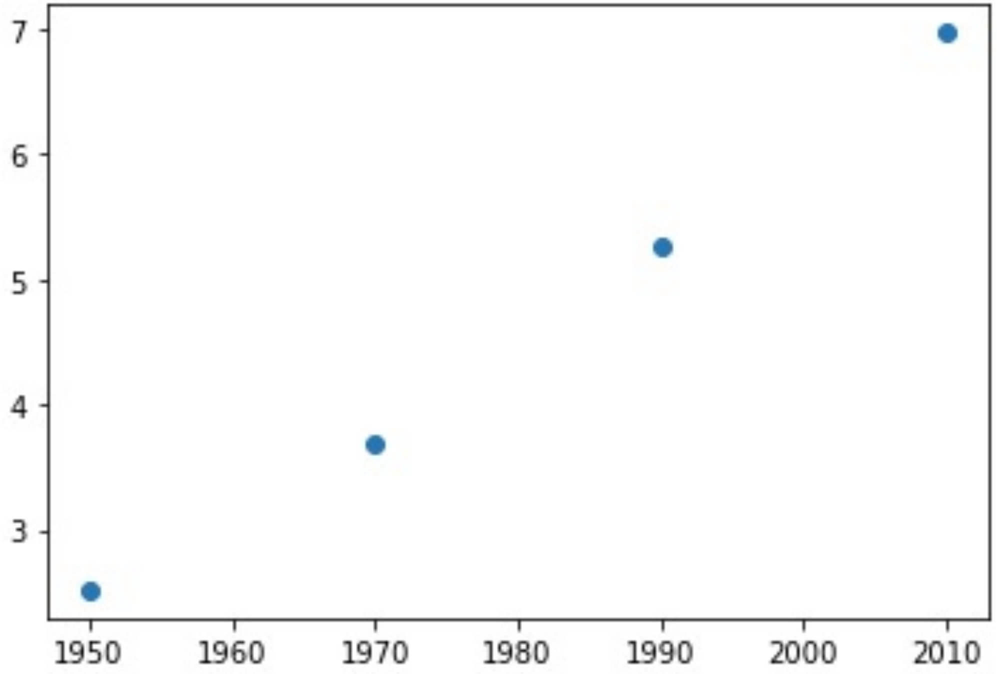
```
year = [1950 , 1970 , 1990 ,  2010]
pop  = [2.519, 3.692, 5.263, 6.972]
```

plt.scatter() :



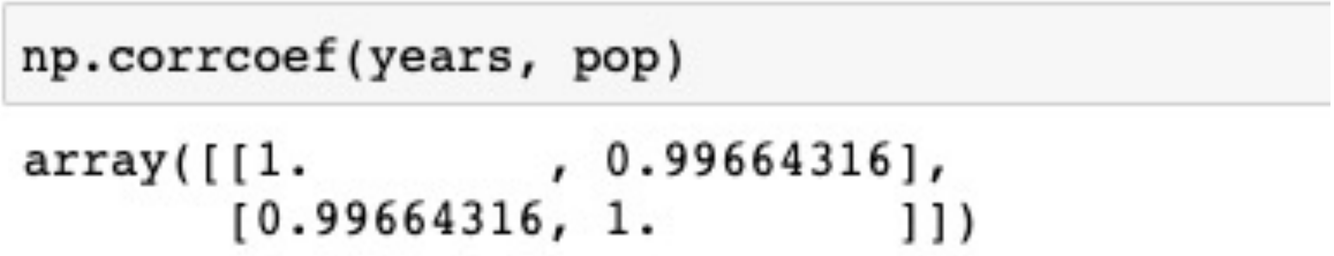
```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

```
plt.scatter(years , pop)
plt.show()
```



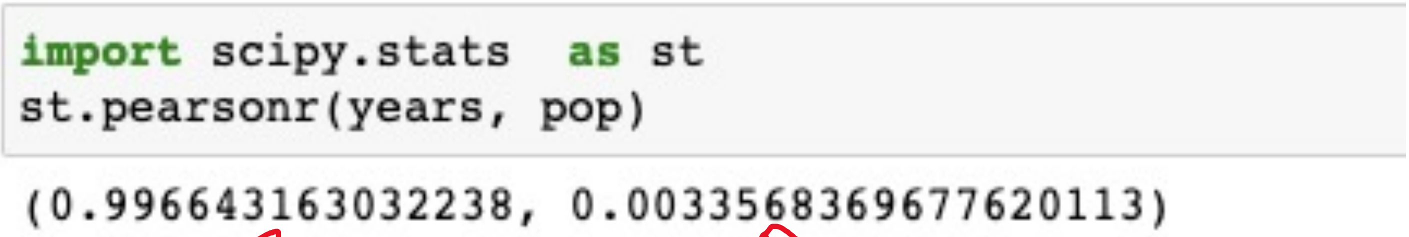
Scatter plot is used when we need to measure the correlation between two attributes.

plt.scatter() :



```
np.corrcoef(years, pop)
```

```
array([[1.        , 0.99664316],
       [0.99664316, 1.        ]])
```



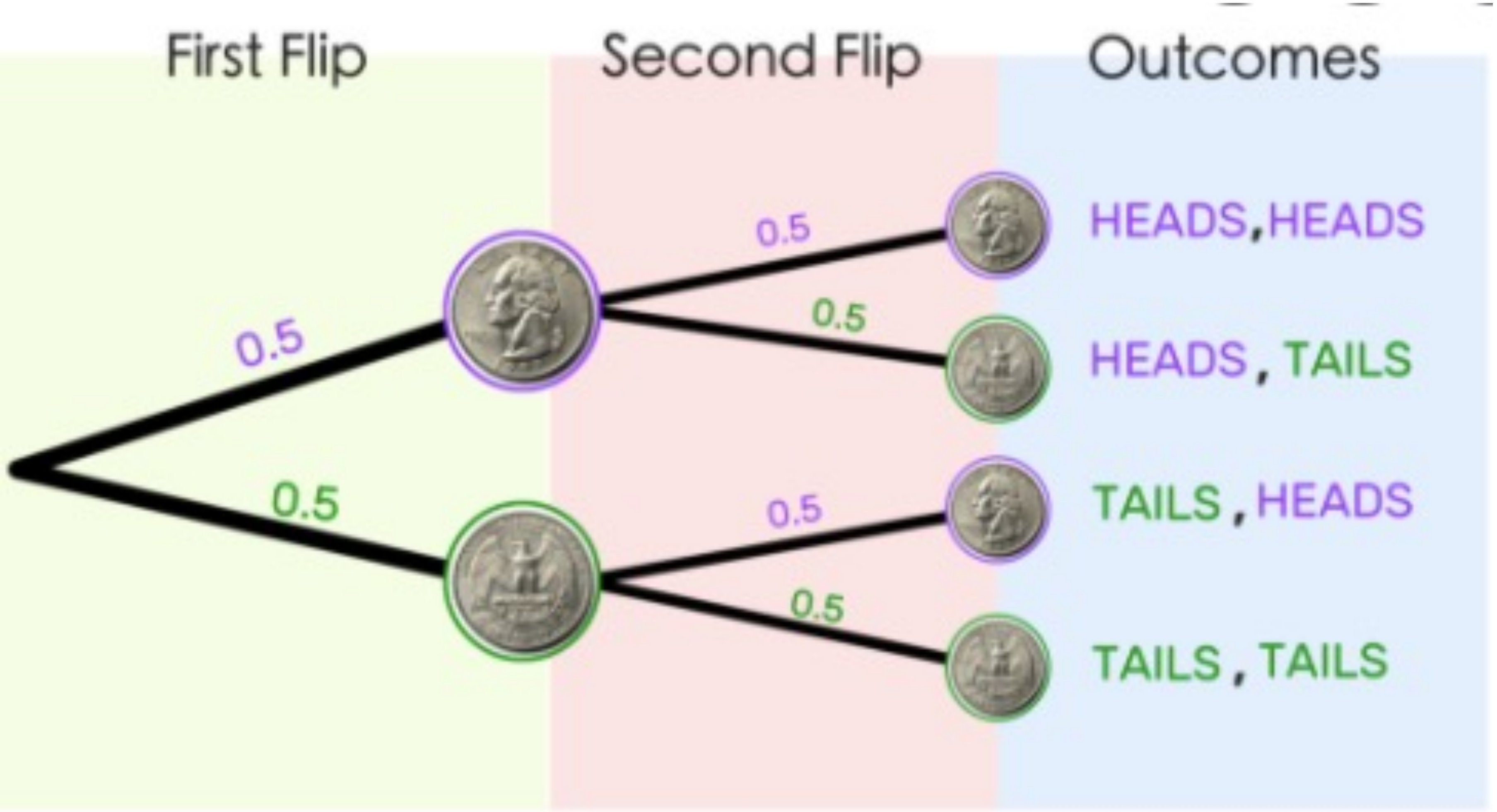
```
import scipy.stats  as st
st.pearsonr(years, pop)
```

```
(0.996643163032238, 0.0033568369677620113)
```

Correlation

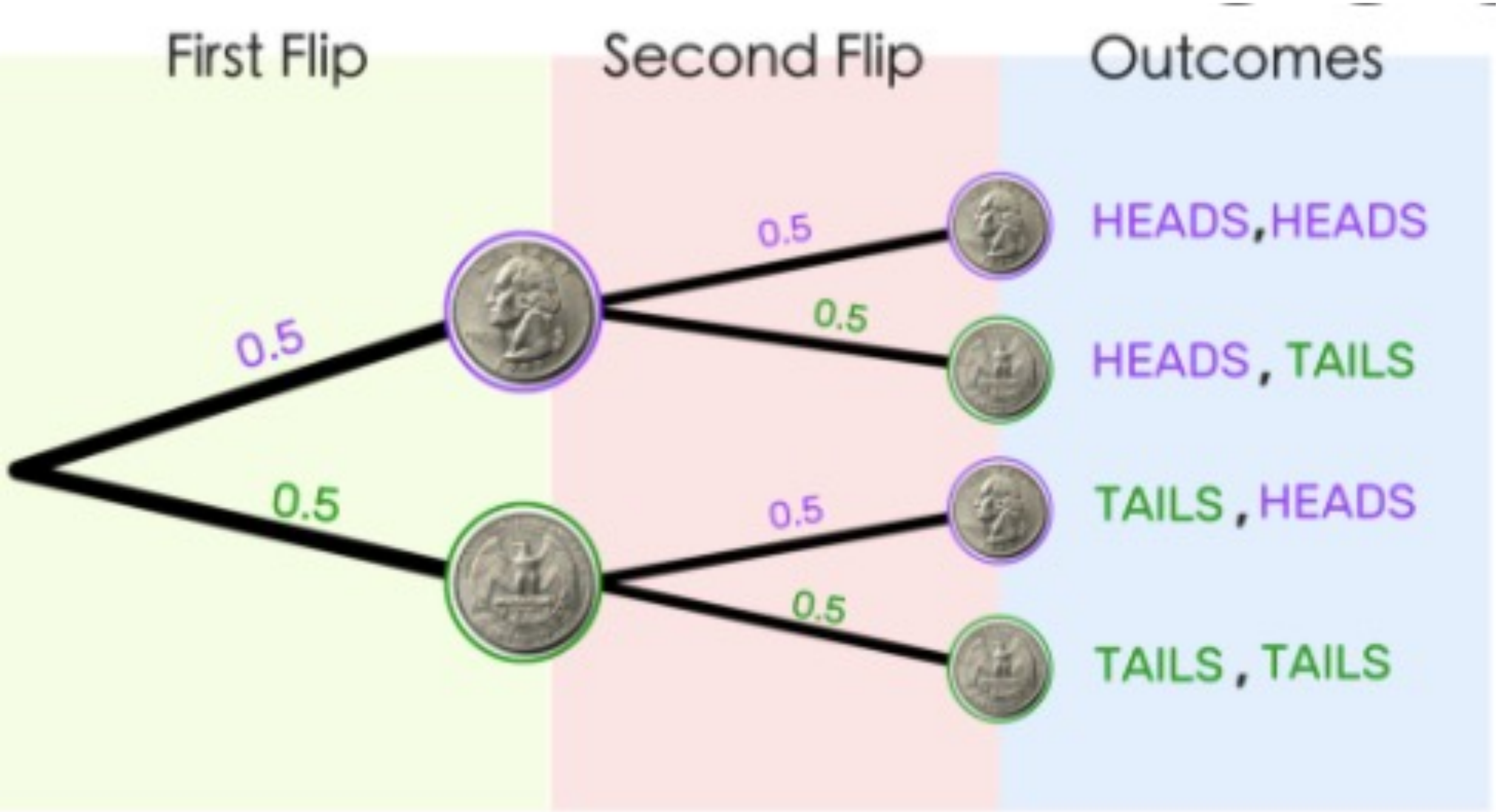
P-value

P-Value:



Probabilities?

P-Value:



$\frac{1}{4}$

$\frac{1}{2}$ one T one H

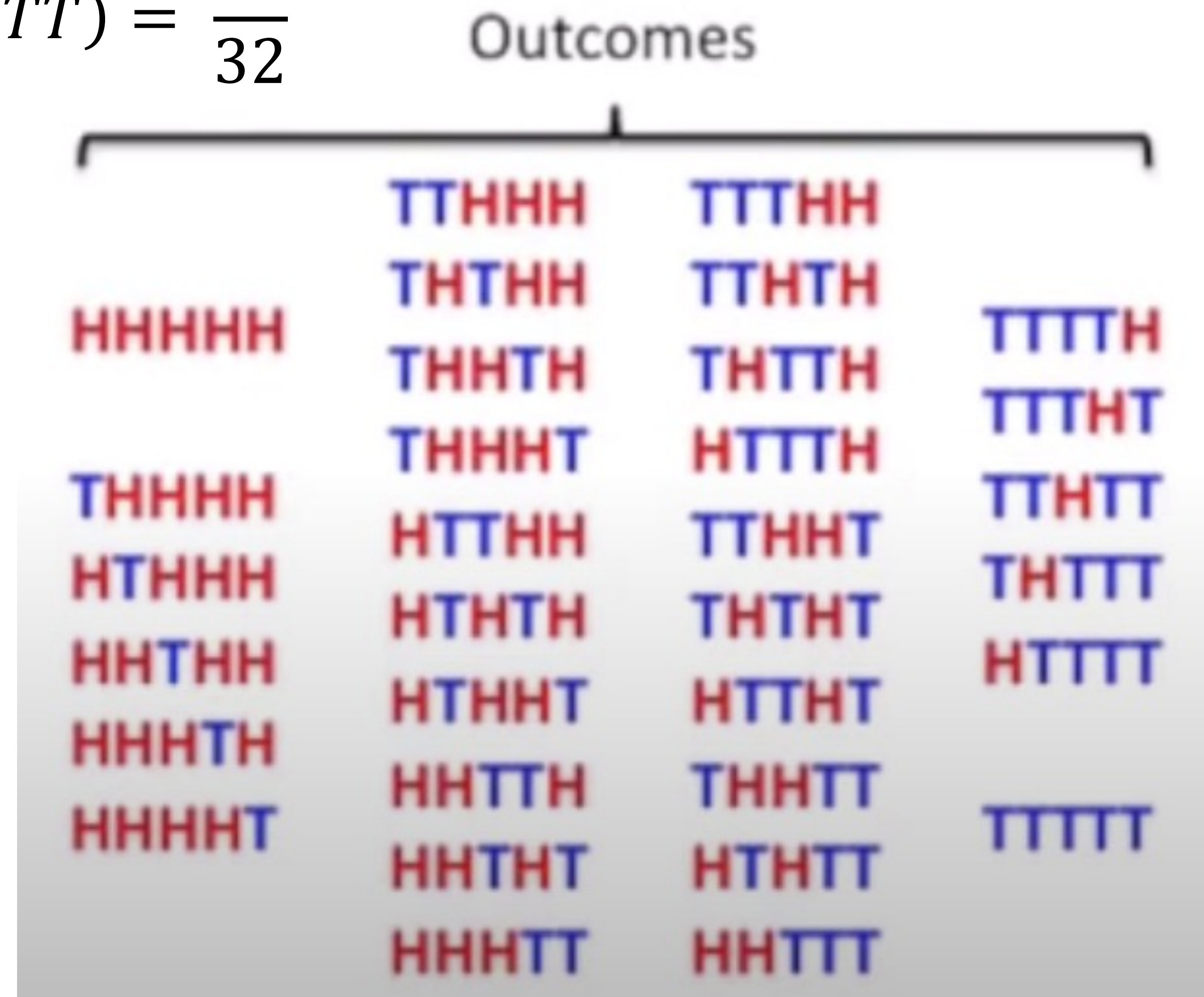
$\frac{1}{4}$

P-Value (flipping a coin 5 times):

$$p(HHHHH) = \frac{1}{32}$$

$$p(TTTTT) = \frac{1}{32}$$

$$p - value(HHHHH) = \frac{1}{32}+\frac{1}{32} =$$
$$0.0625$$



Outcomes

HHHHH

THHHH
HTHHH
HHTHH
HHHTH
HHHHT

TTHHH
THTHH
THHTH
THHHT
HTTHH
HTHTH
HTHHT
HHTTH
HHTHT
HHHTT

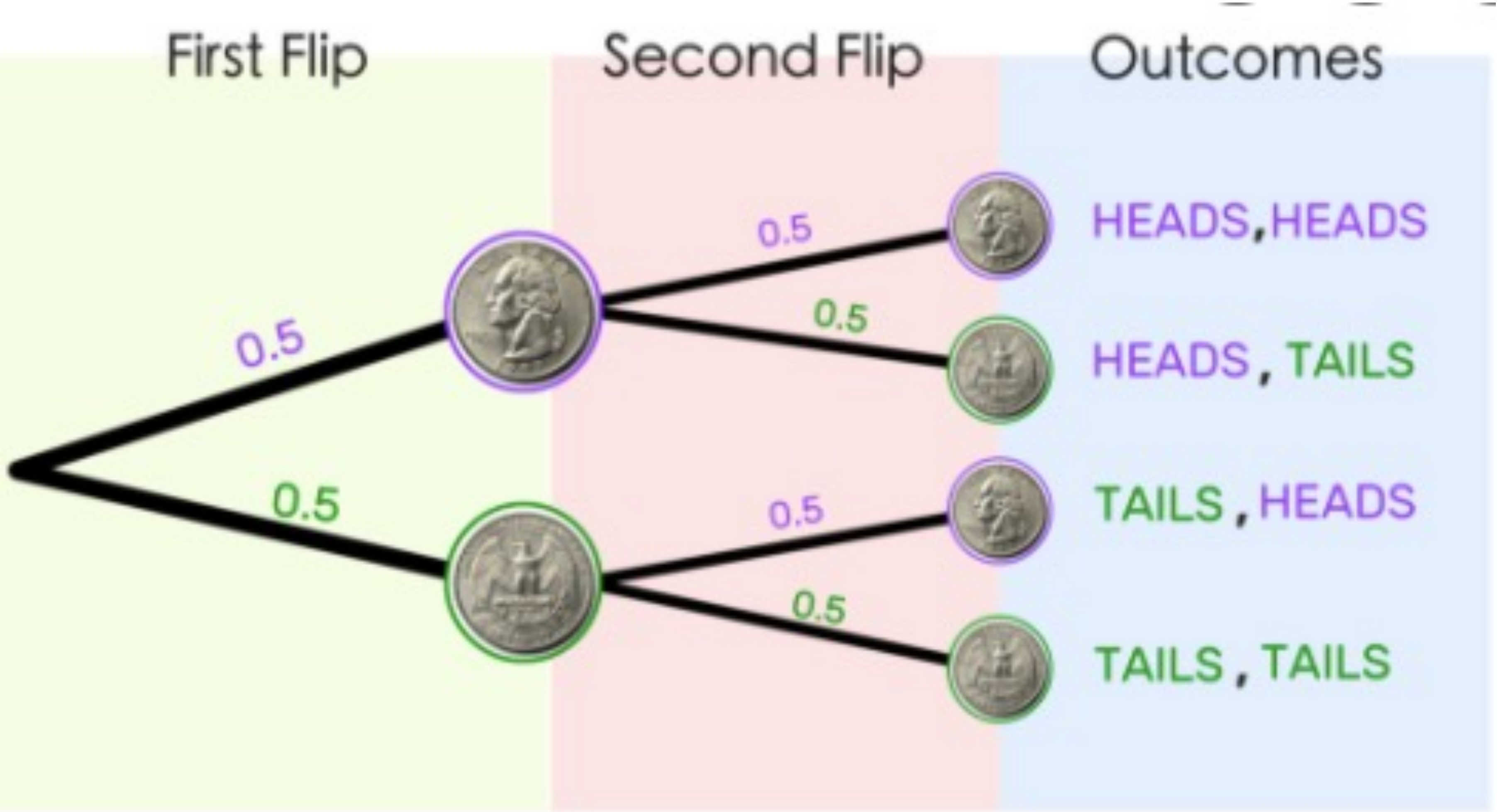
TTTHH
TTHTH
THTTH
HTTTH
TTHHT
THTHT
HTTHT
THHTT
HTHTT
HHTTT

TTTTH
TTTHT
TTHTT
THTTT
HTTTT

TTTTT

P-Value:

probability that random chance generated the data or somethig else that is equal or rarer.



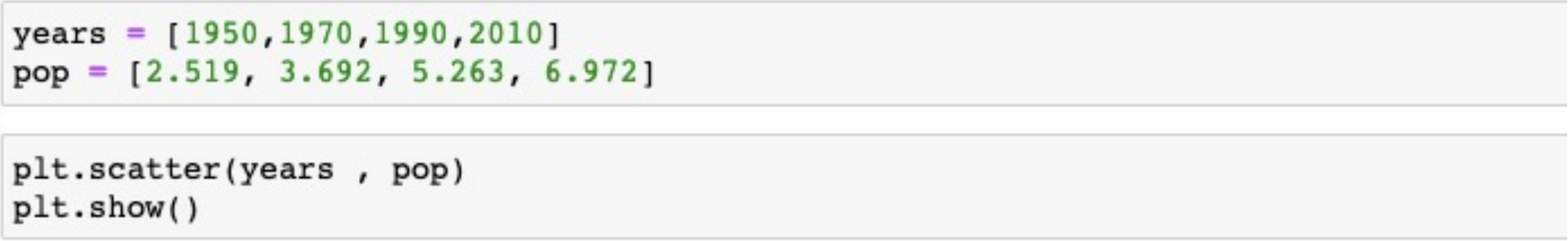
P-values:

$$\frac{1}{4}+\frac{1}{4}=\frac{1}{2}$$

$$1$$

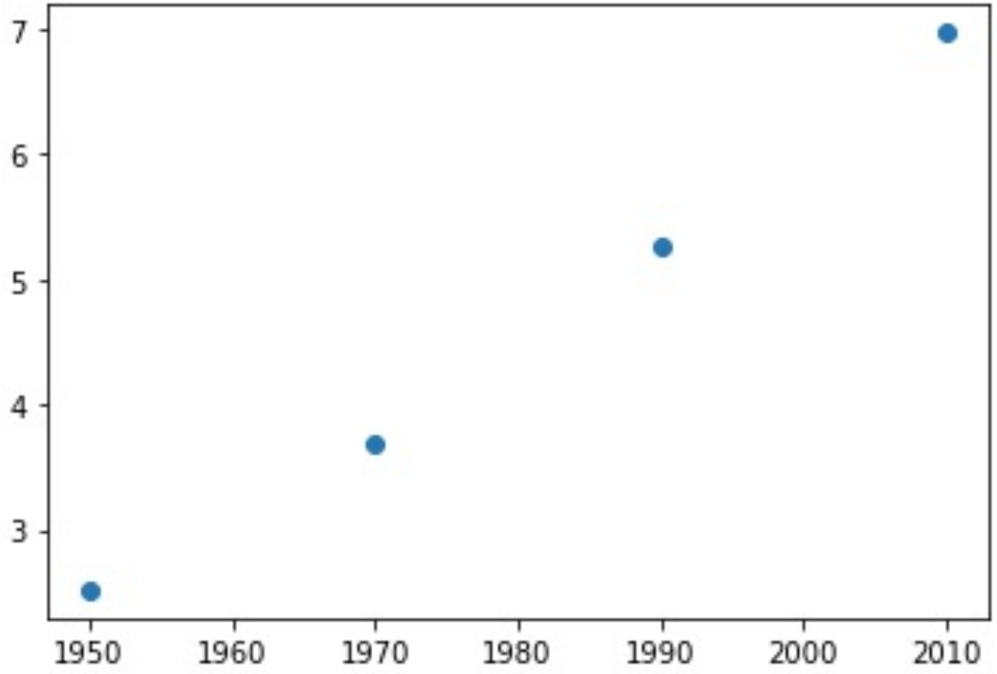
$$\frac{1}{2}$$

plt.scatter() :



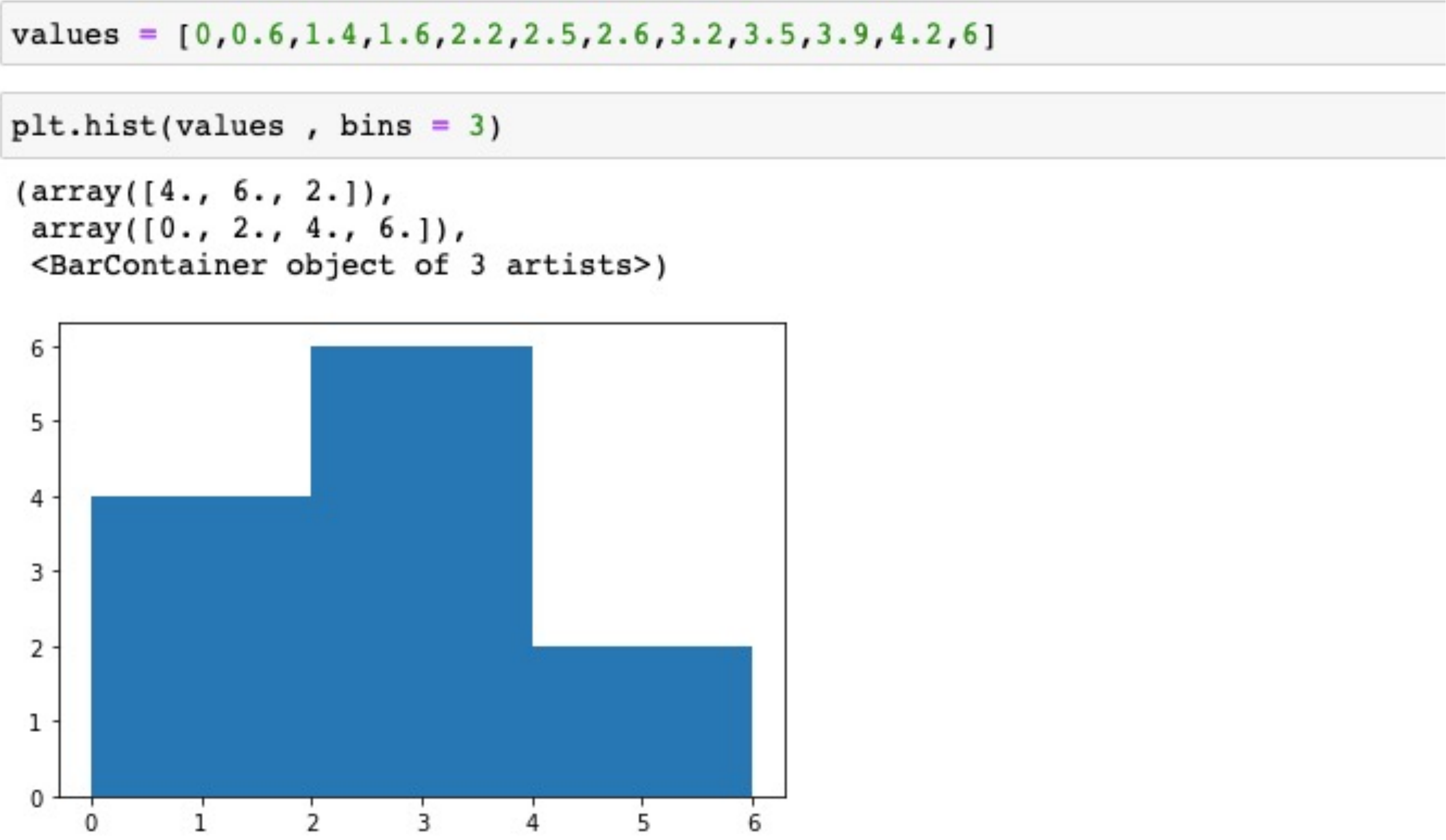
```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
```

```
plt.scatter(years , pop)
plt.show()
```



Scatter plot is used when we need to measure the correlation between two attributes.

plt.hist() :

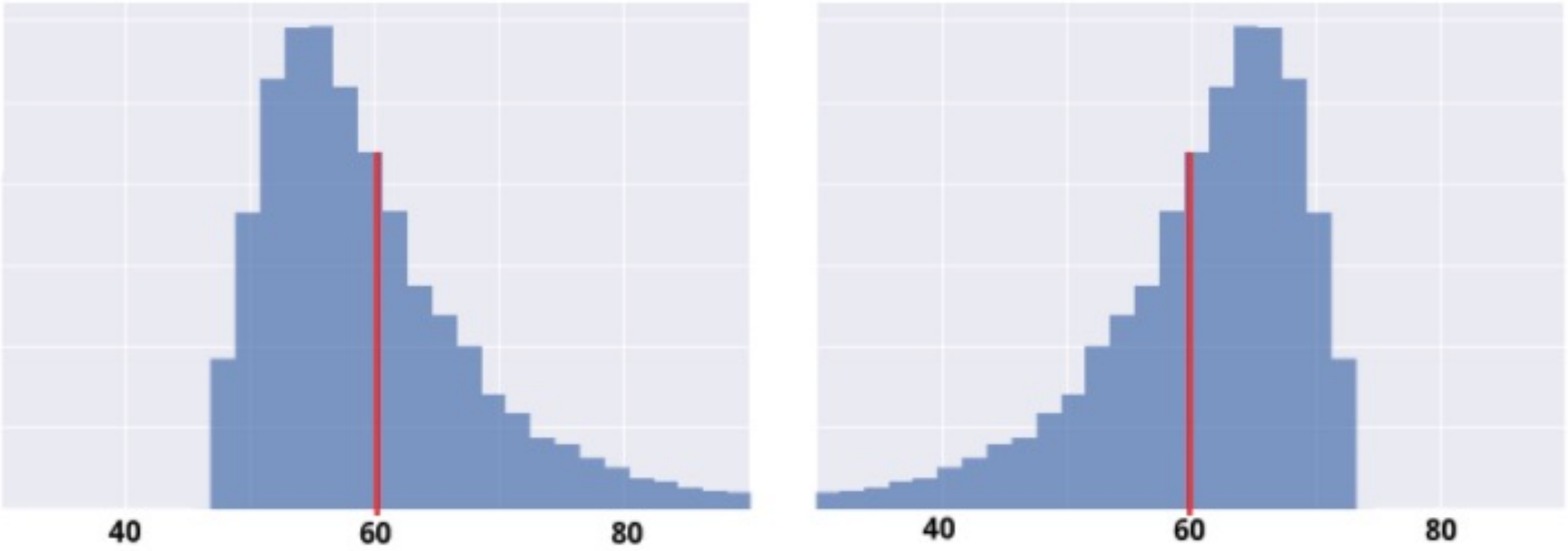


```
values = [0,0.6,1.4,1.6,2.2,2.5,2.6,3.2,3.5,3.9,4.2,6]
```

```
plt.hist(values , bins = 3)
```

```
(array([4., 6., 2.]),
 array([0., 2., 4., 6.]),
 <BarContainer object of 3 artists>)
```

plt.hist() :



Distribution of Data:

- Which is the most frequent data? statistics.mode()
- The data is centered around which point? Nupmy.mean()
- What is the value observed in 50% of the time? Numpy.median()
- How vary the values are ? np.std()

Distribution of Data:

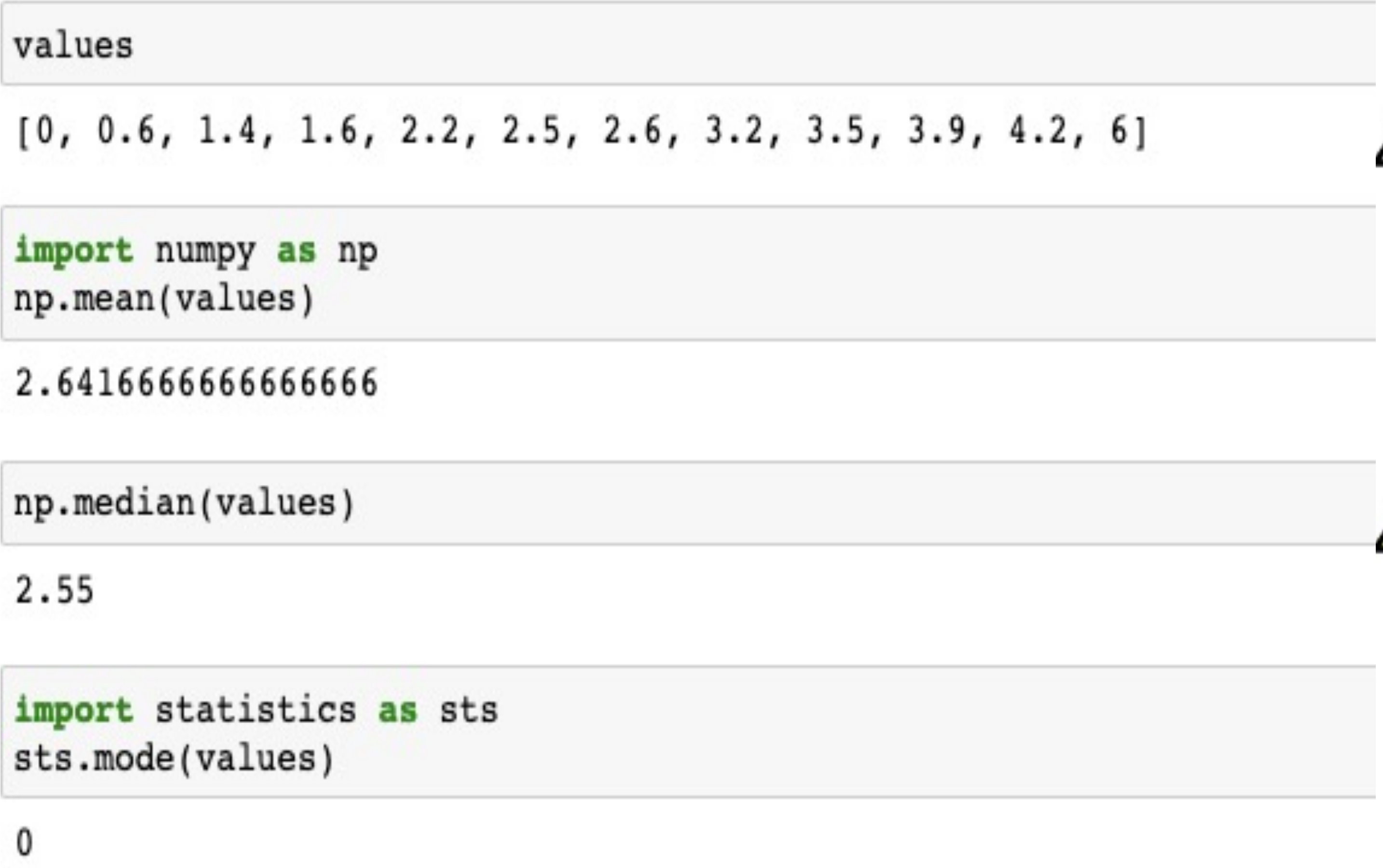
Most of the time it takes 80 mins

Half of the times it takes 80 mins

On average it takes 80 mins

How long does it take
to go from
City A to city B

Distribution of Data:



```
values
```

```
[0, 0.6, 1.4, 1.6, 2.2, 2.5, 2.6, 3.2, 3.5, 3.9, 4.2, 6]
```

```
import numpy as np
np.mean(values)
```

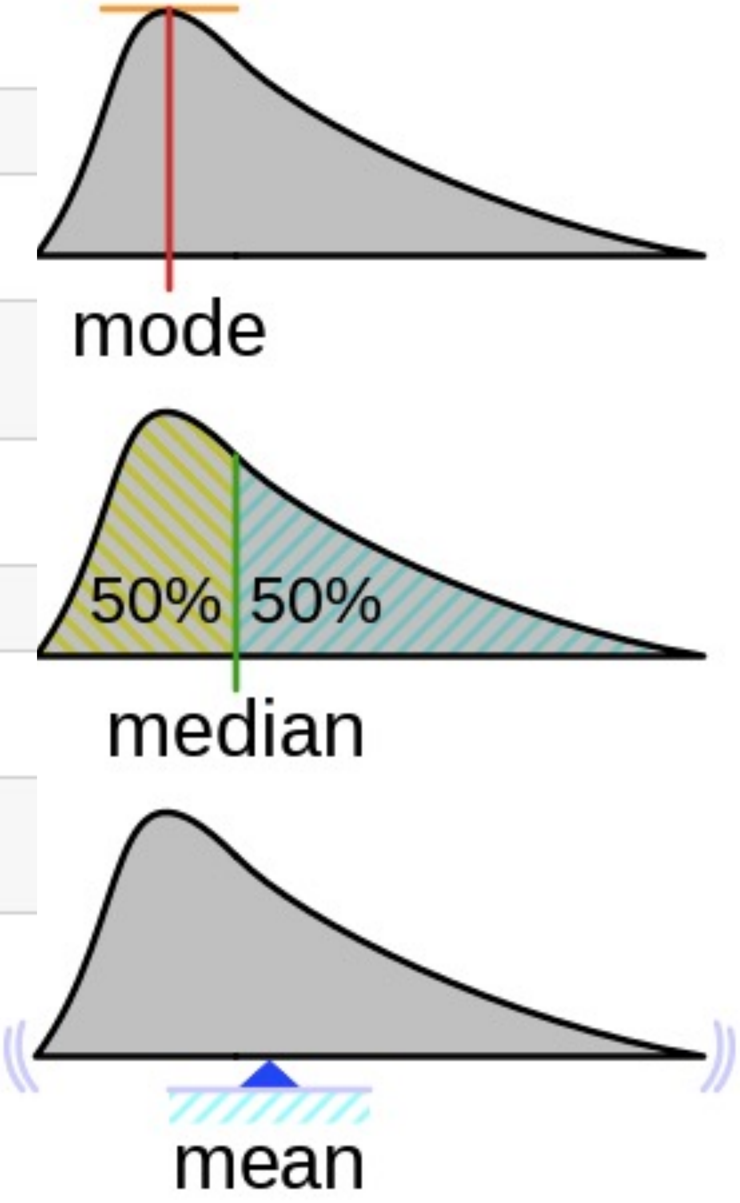
```
2.6416666666666666
```

```
np.median(values)
```

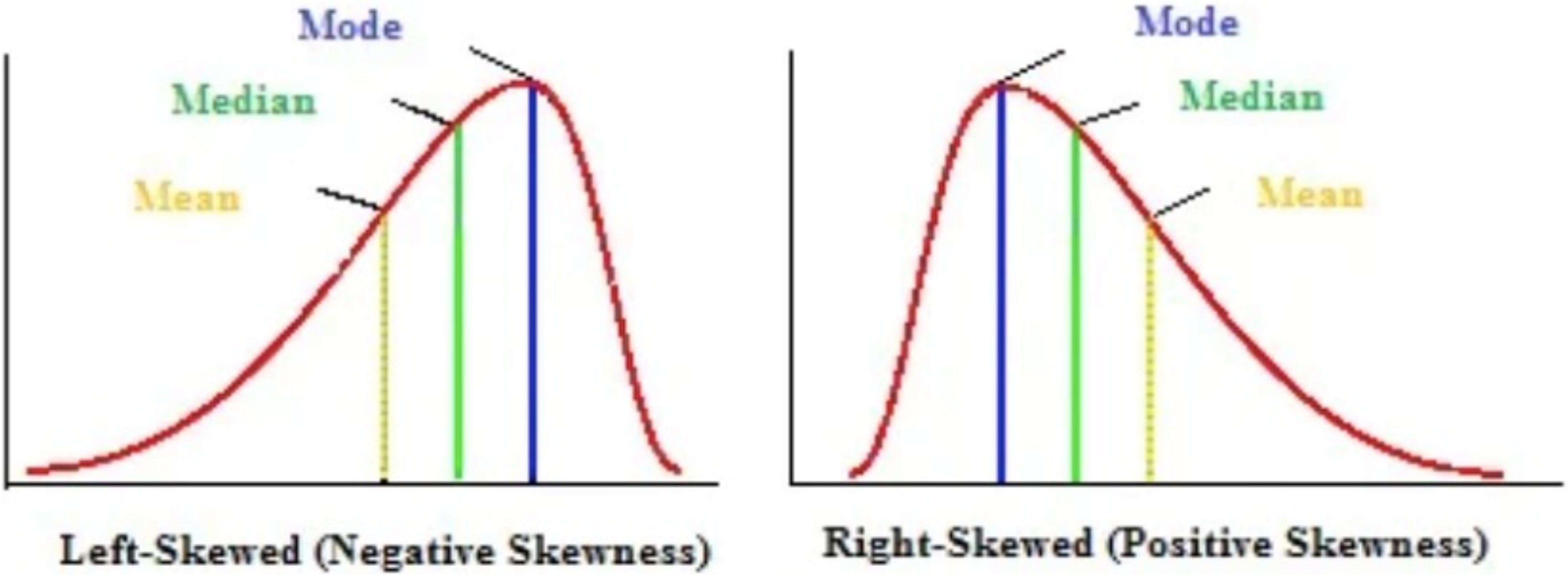
```
2.55
```

```
import statistics as sts
sts.mode(values)
```

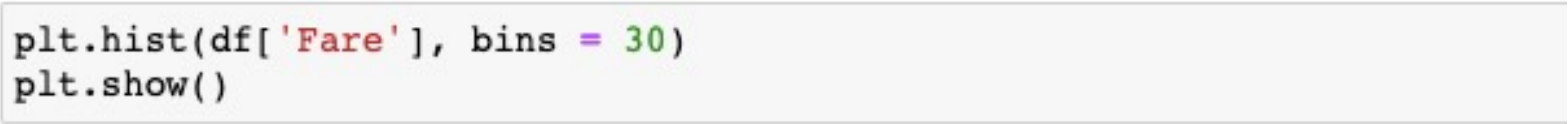
```
0
```



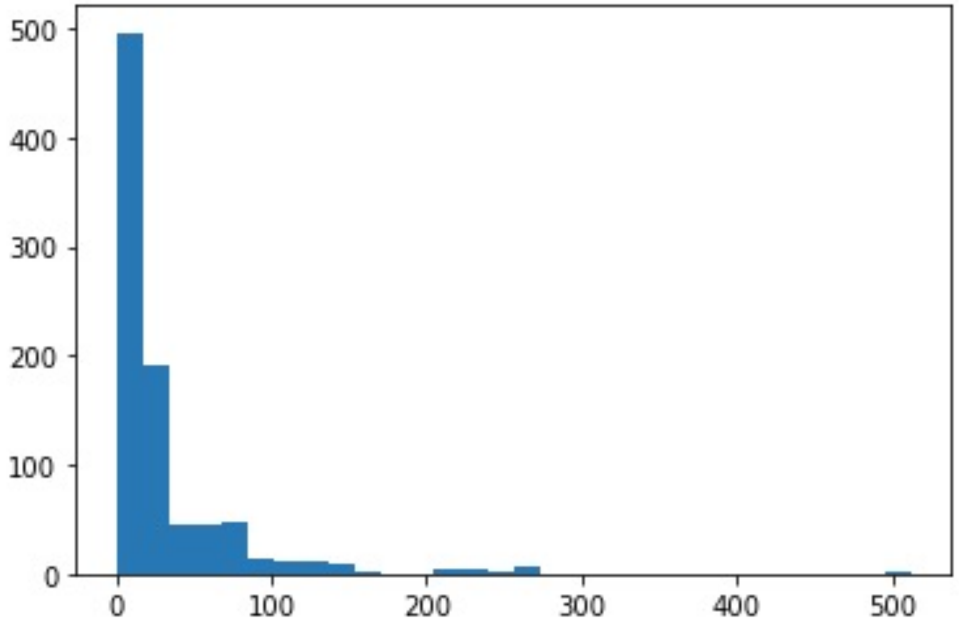
Distribution of Data:

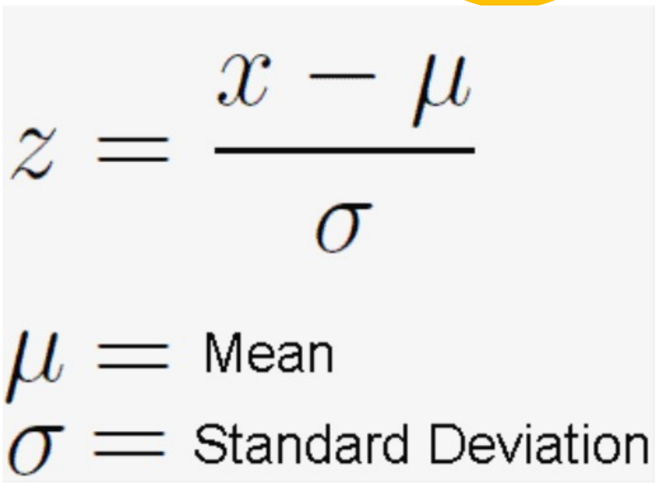

Mode
Median
Mean
Left-Skewed (Negative Skewness)
Mode
Median
Mean
Right-Skewed (Positive Skewness)

Distribution of Data:



```
plt.hist(df['Fare'], bins = 30)
plt.show()
```

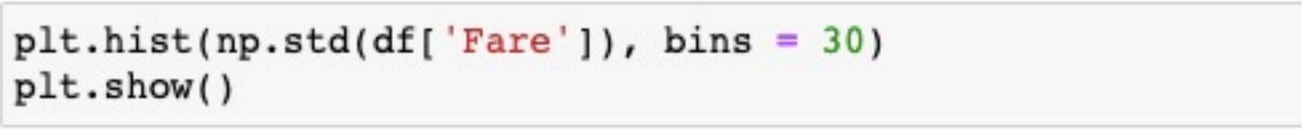




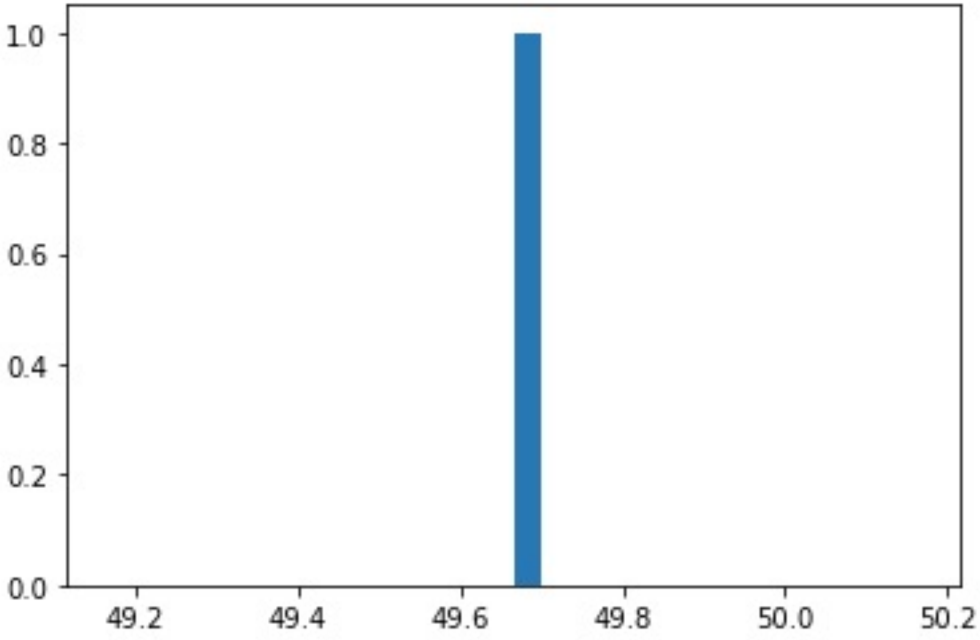
$$z = \frac{x - \mu}{\sigma}$$

μ = Mean

σ = Standard Deviation



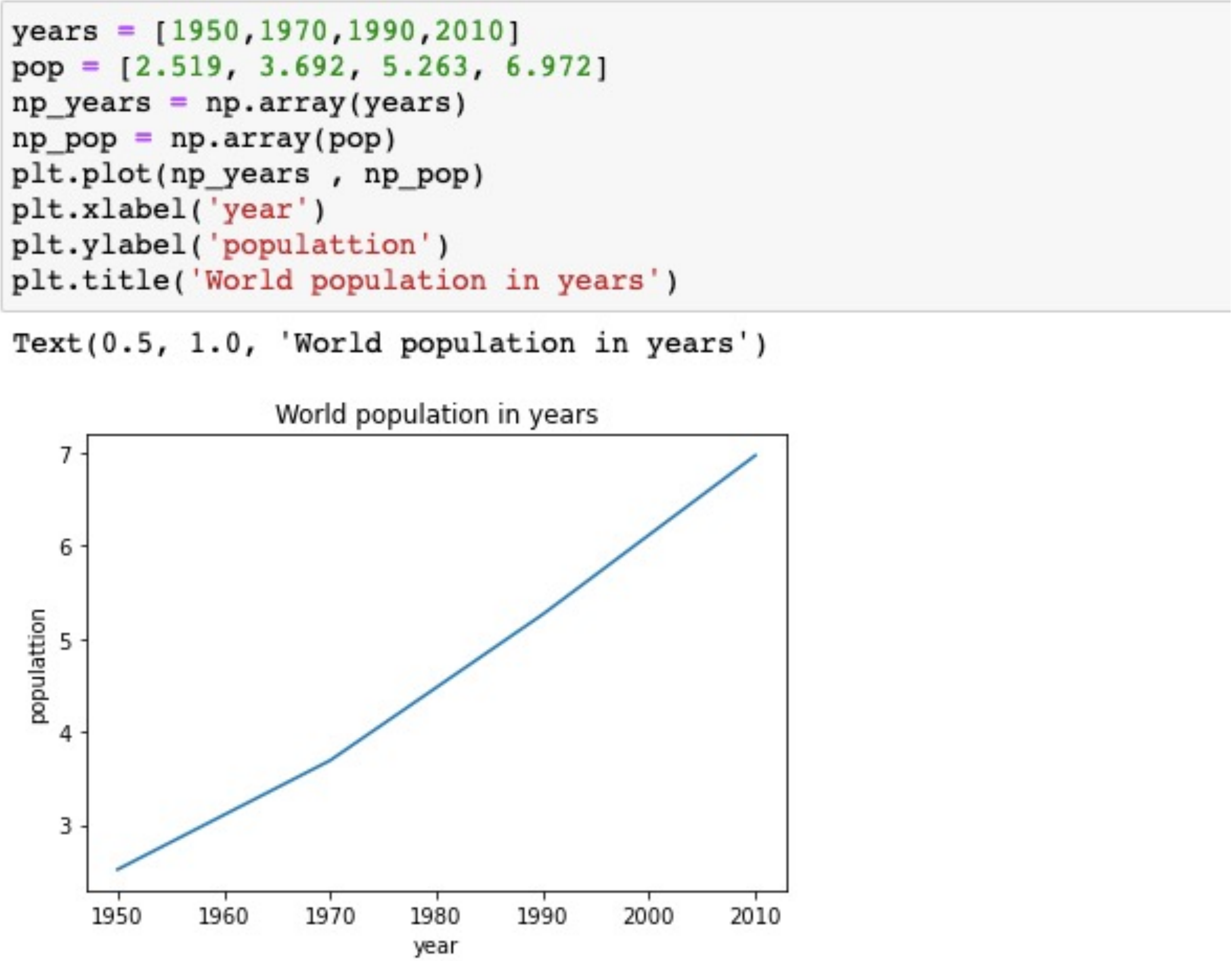
```
plt.hist(np.std(df['Fare']), bins = 30)
plt.show()
```



Customization:

- Add labels to the axis: plt.xlabel() , plt.ylabel()
- Add Title to the plot : plt.title()
- Changing values one the axis: plt.xticks() , plt.yticks()
- Labeling values on the axis

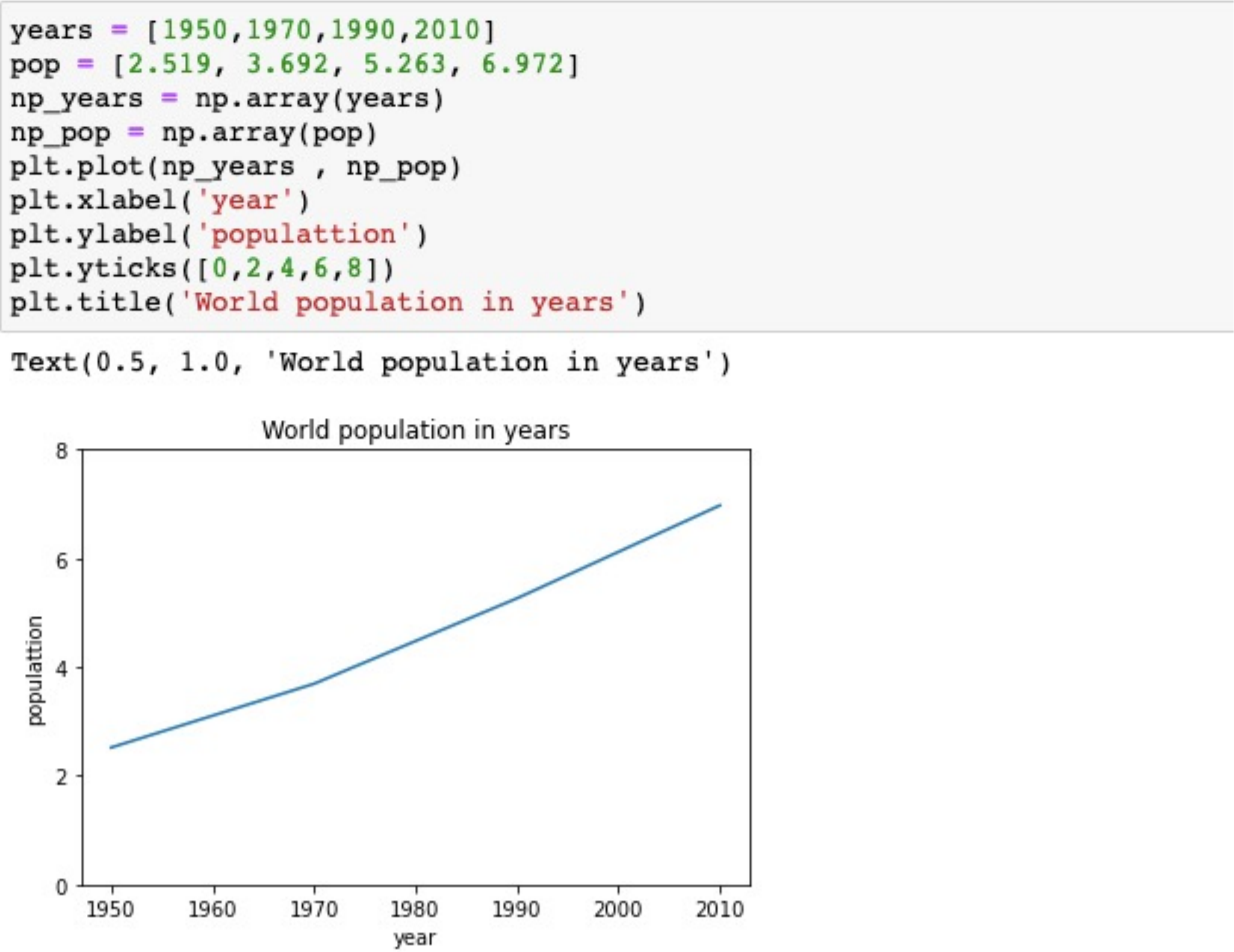
Customization:



```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.title('World population in years')
```

```
Text(0.5, 1.0, 'World population in years')
```

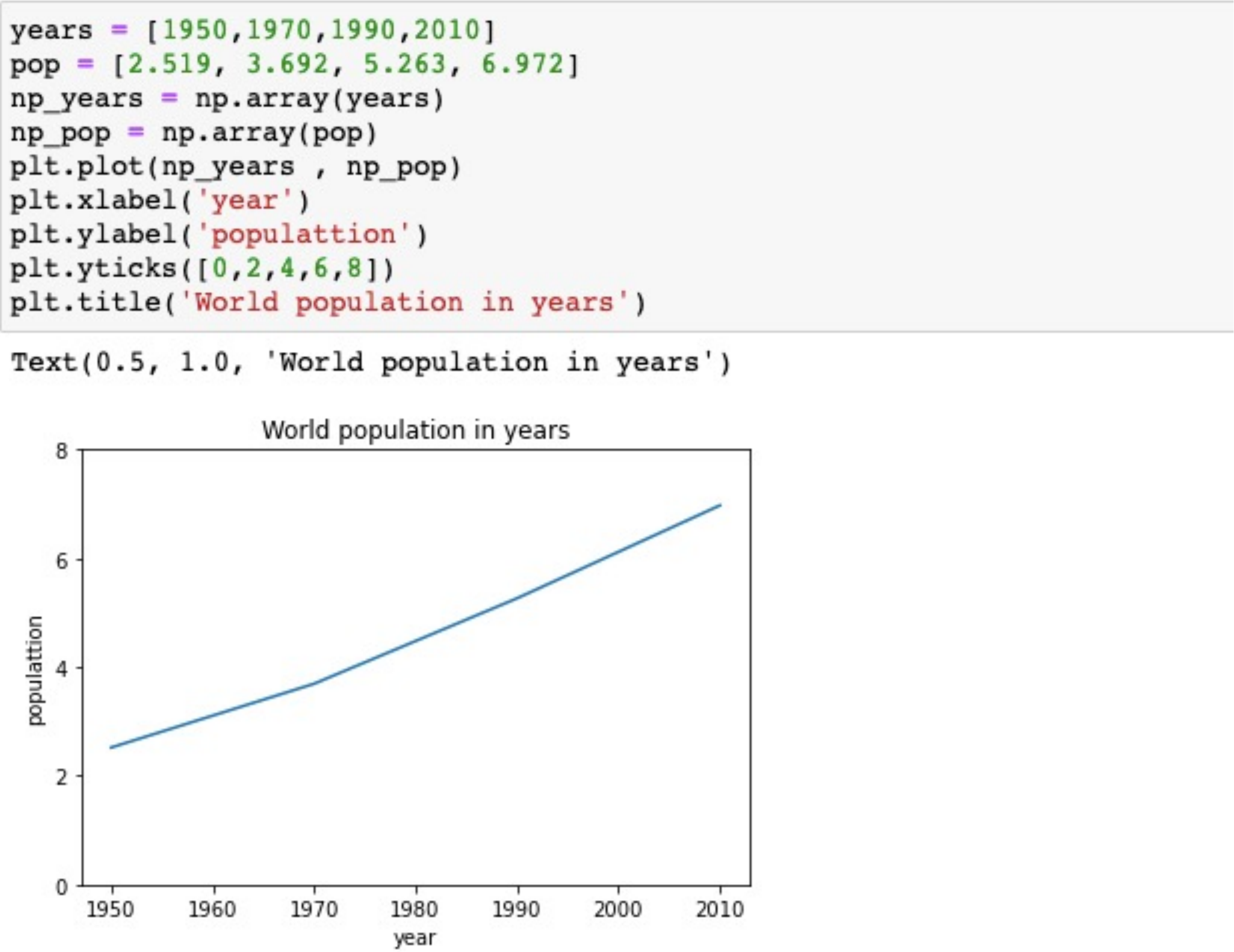
Customization:



```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.yticks([0,2,4,6,8])
plt.title('World population in years')
```

```
Text(0.5, 1.0, 'World population in years')
```

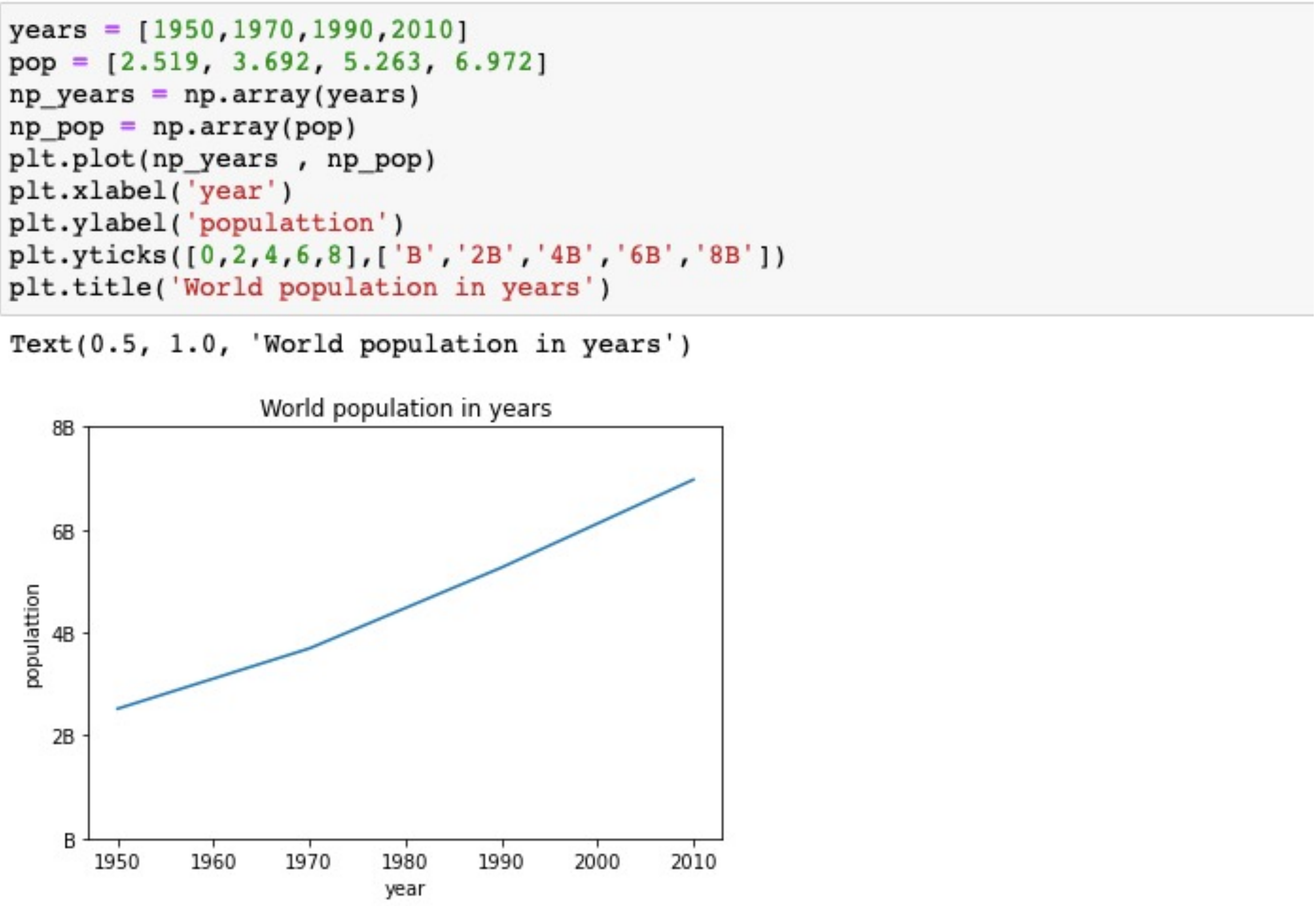
Customization:



```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.yticks([0,2,4,6,8])
plt.title('World population in years')
```

```
Text(0.5, 1.0, 'World population in years')
```

Customization:



```
years = [1950,1970,1990,2010]
pop = [2.519, 3.692, 5.263, 6.972]
np_years = np.array(years)
np_pop = np.array(pop)
plt.plot(np_years , np_pop)
plt.xlabel('year')
plt.ylabel('populattion')
plt.yticks([0,2,4,6,8],['B','2B','4B','6B','8B'])
plt.title('World population in years')
```

```
Text(0.5, 1.0, 'World population in years')
```

Machine Learning:

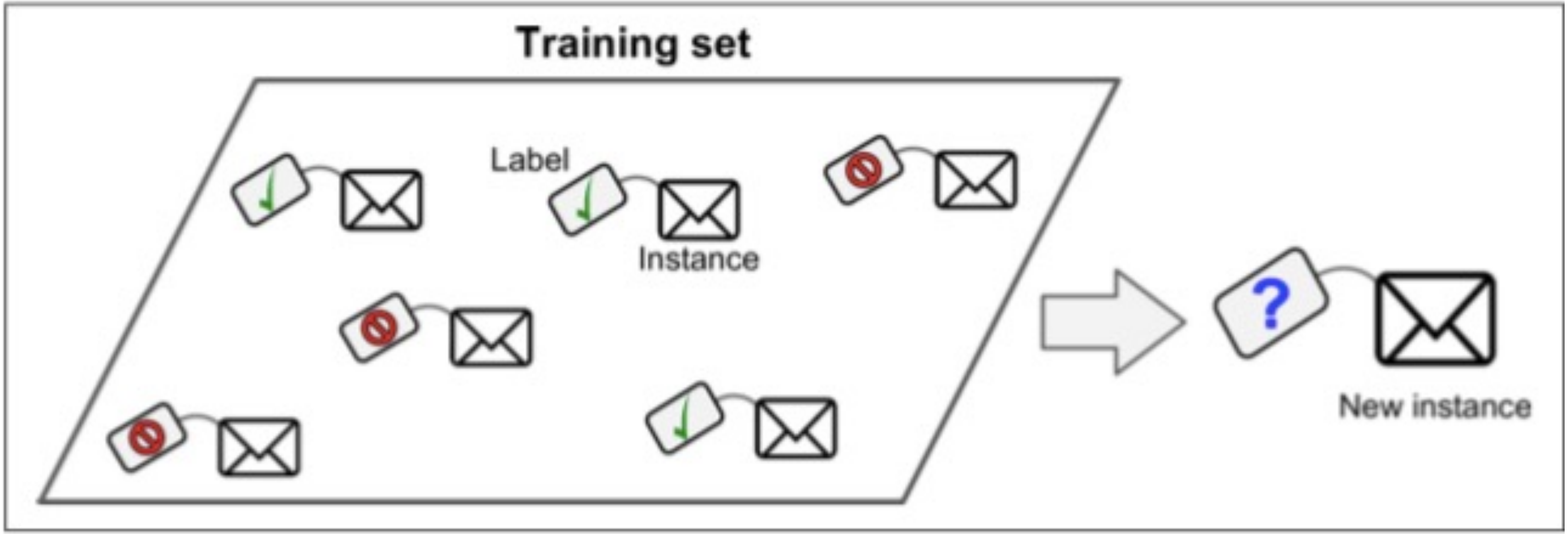
How you write a code with traditional programming technique to detect spams?

- ➢ What a spam looks like, what are the patterns,
- ➢ Write a detection algorithm for each pattern,… .

Problem??

There is an infinite number of patterns!

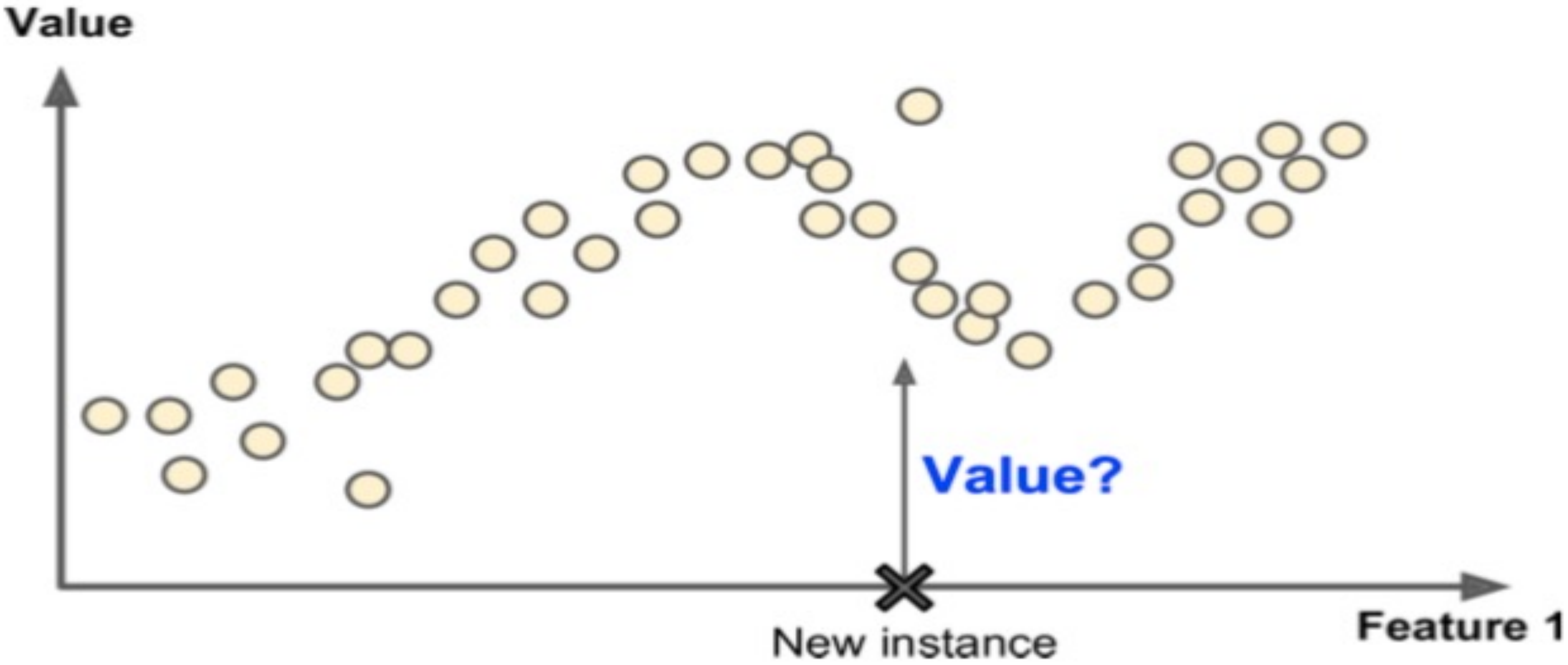
ML (Supervised):



In ML, the model will learn (based on some examples) which patterns are representative of a spam.

Classification

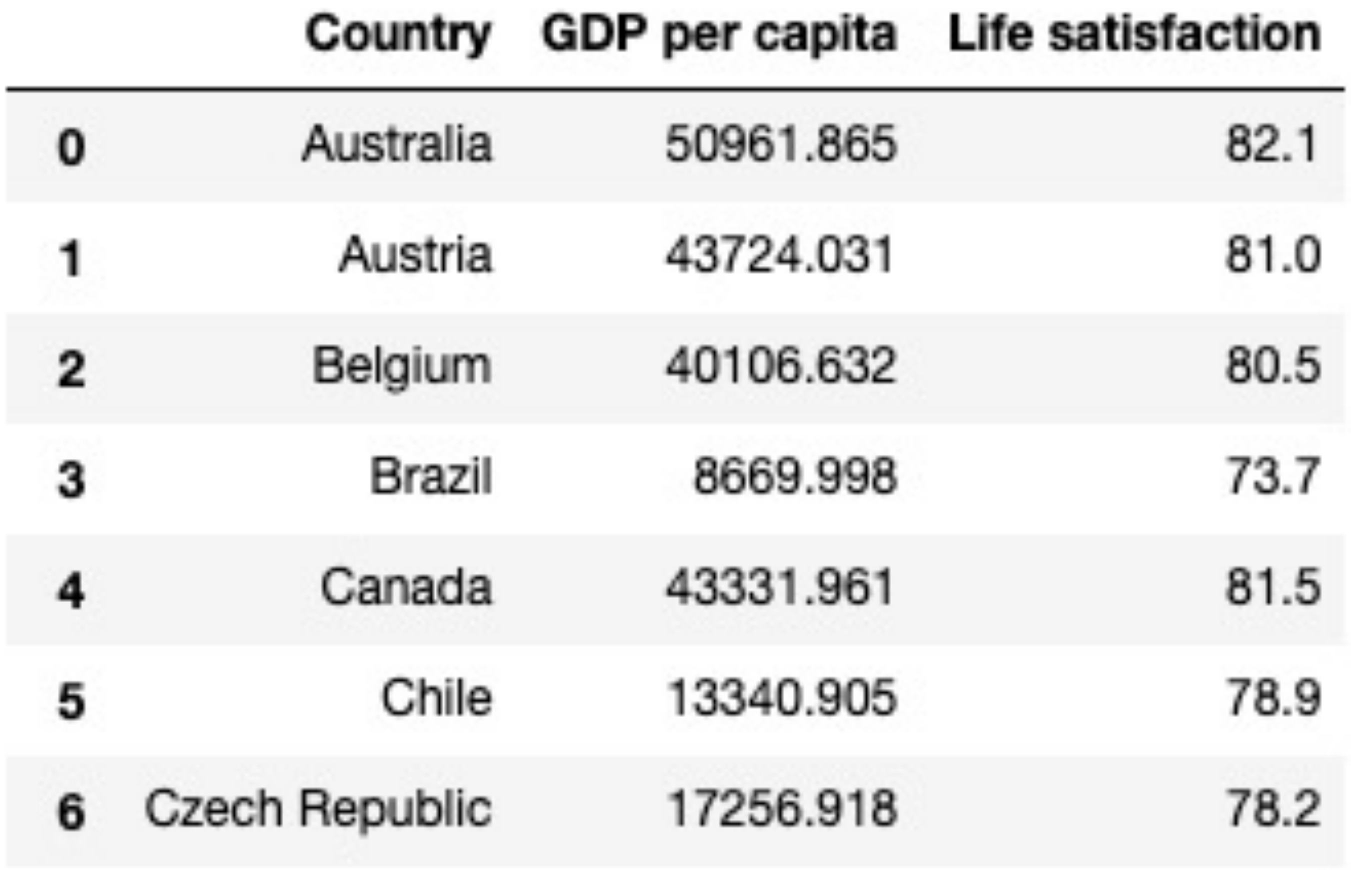
ML (Supervised):



Given a set of Instances and their corresponging value, we can quess what is the value of a newly entered instance.

Regression

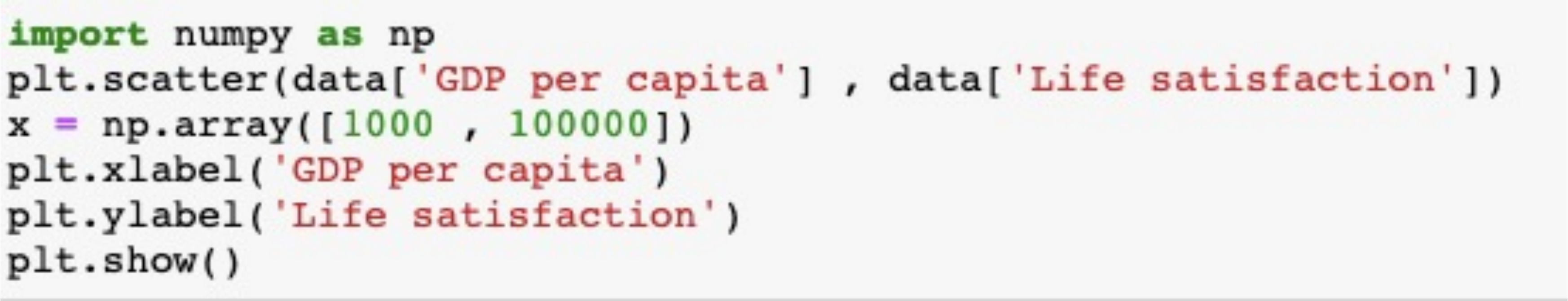
ML (Example):



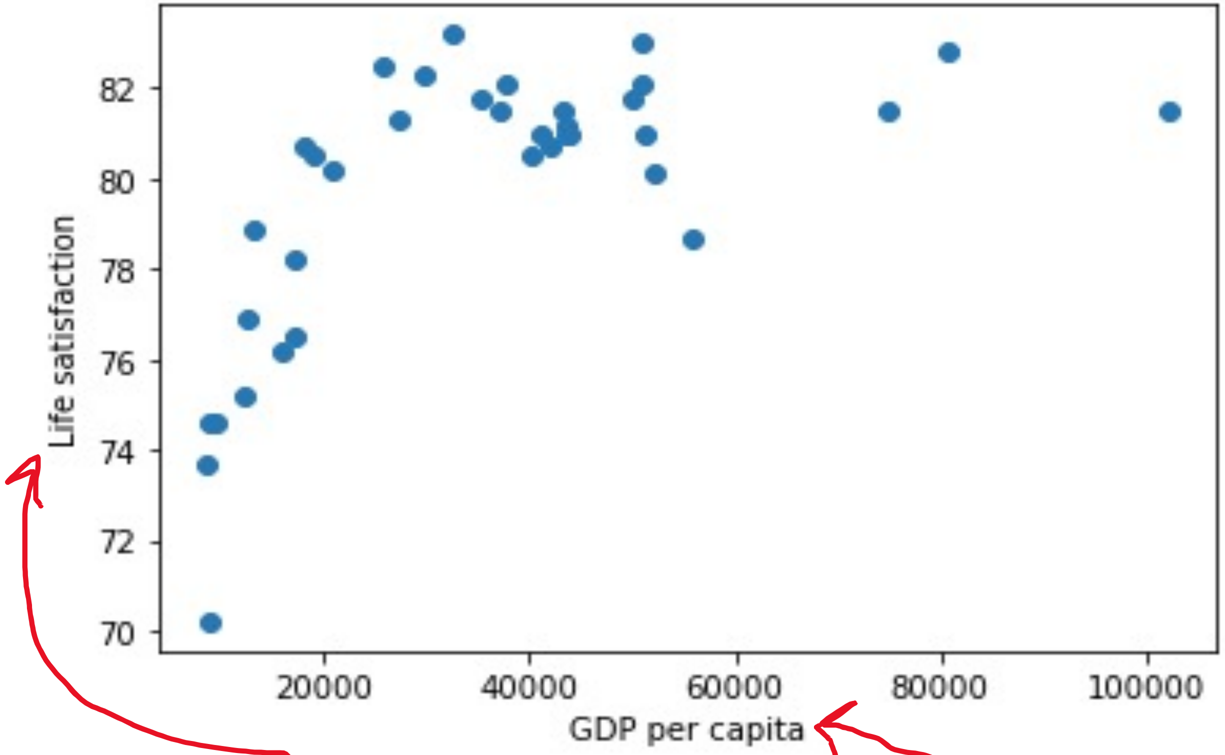
	Country	GDP per capita	Life satisfaction
0	Australia	50961.865	82.1
1	Austria	43724.031	81.0
2	Belgium	40106.632	80.5
3	Brazil	8669.998	73.7
4	Canada	43331.961	81.5
5	Chile	13340.905	78.9
6	Czech Republic	17256.918	78.2

Given a GDP per capita in a country, can you guess what is the life satisfaction index?

ML (Example):



```
import numpy as np
plt.scatter(data['GDP per capita'] , data['Life satisfaction'])
x = np.array([1000 , 100000])
plt.xlabel('GDP per capita')
plt.ylabel('Life satisfaction')
plt.show()
```



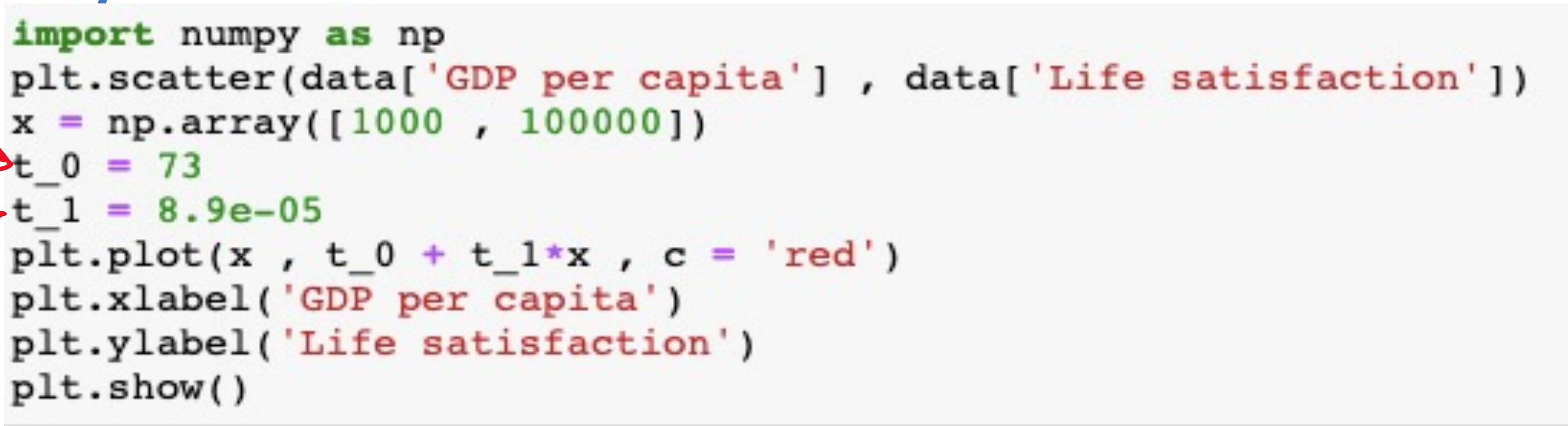
What is the simplest and common pattern in the scatter plot?

$$y = \theta_0 + \theta_1 x$$

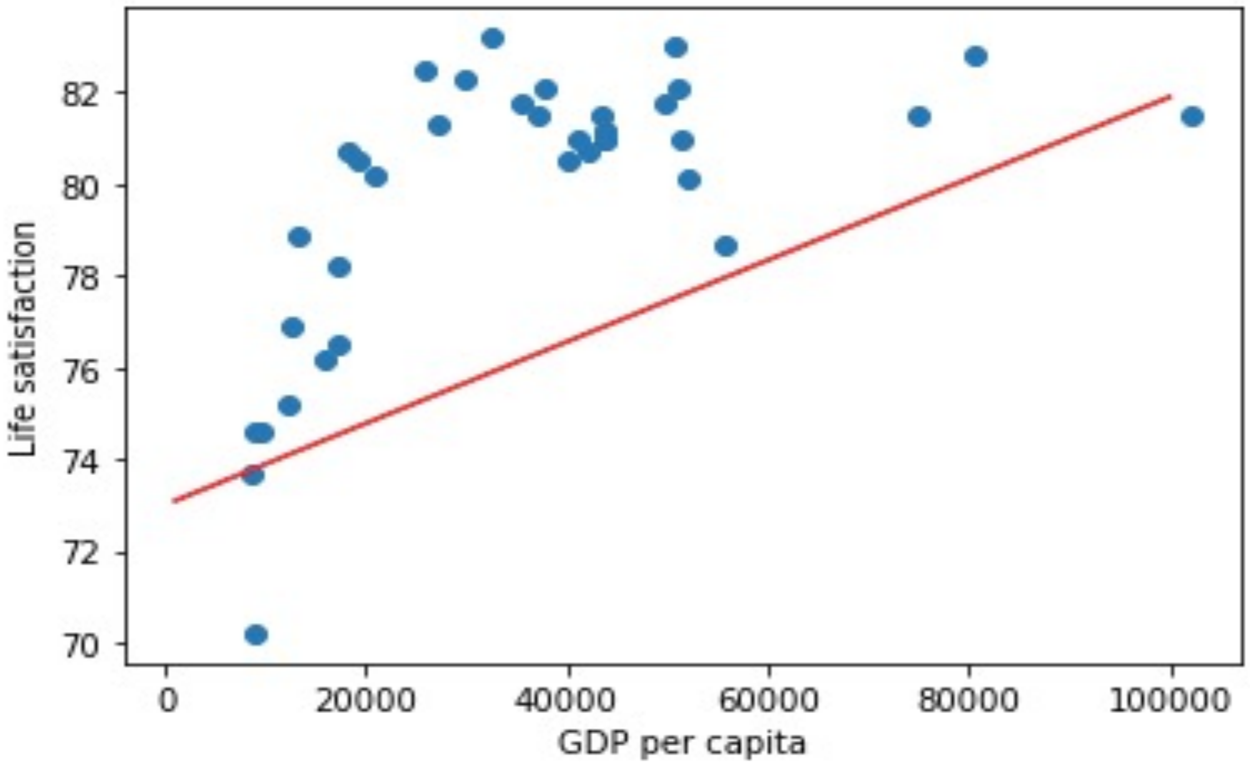
ML (Example):

θ_0

θ_1



```
import numpy as np
plt.scatter(data['GDP per capita'] , data['Life satisfaction'])
x = np.array([1000 , 100000])
t_0 = 73
t_1 = 8.9e-05
plt.plot(x , t_0 + t_1*x , c = 'red')
plt.xlabel('GDP per capita')
plt.ylabel('Life satisfaction')
plt.show()
```

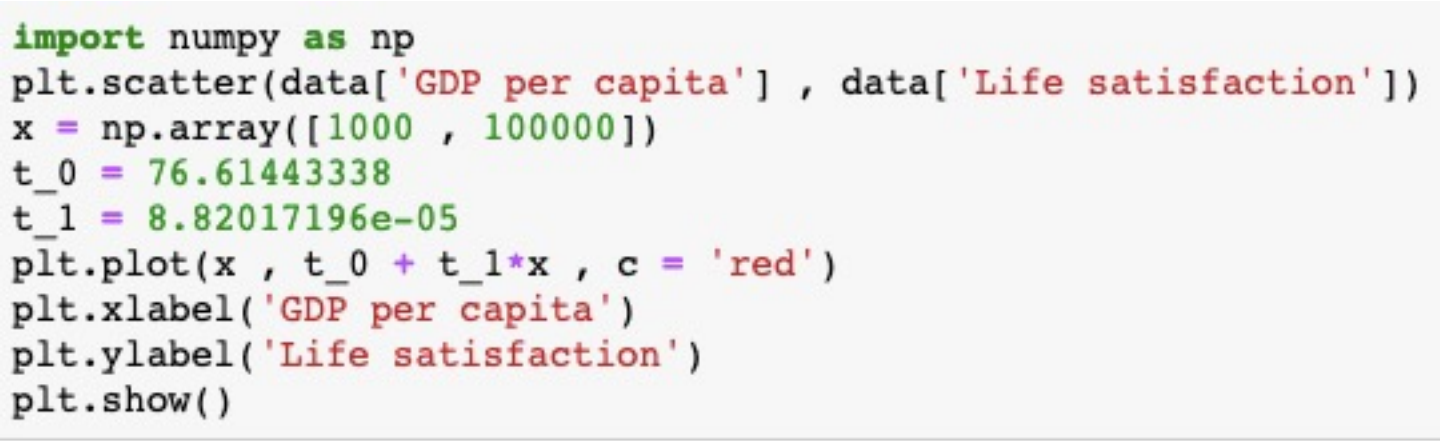


$$life\ satisfaction = \theta_0 + \theta_1 * \ GDP\ per\ Capita$$

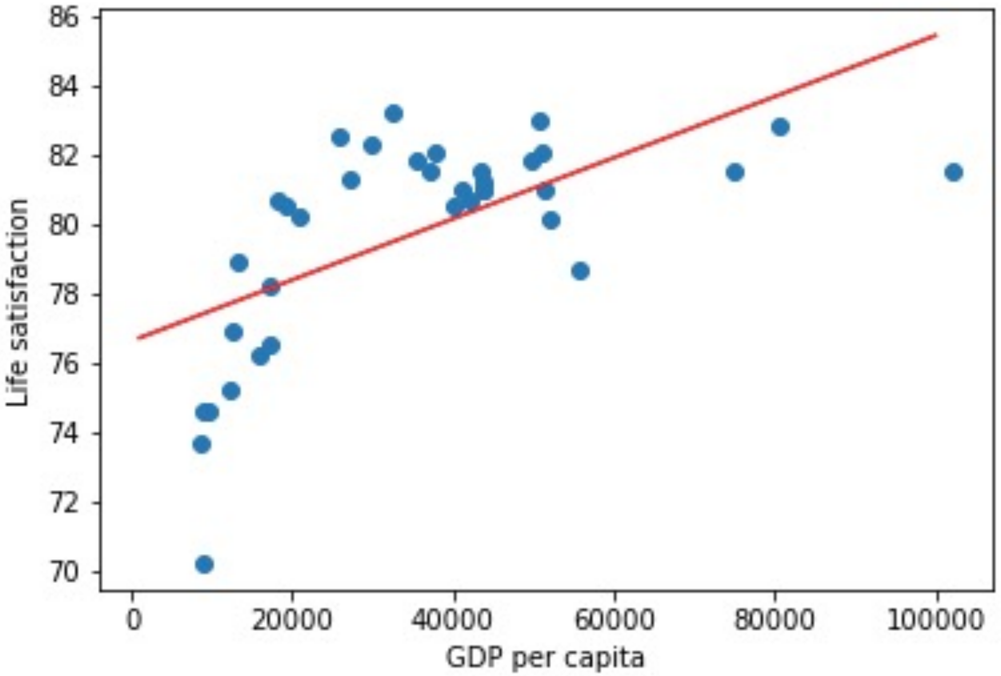
ML (Example):

θ_0

θ_1

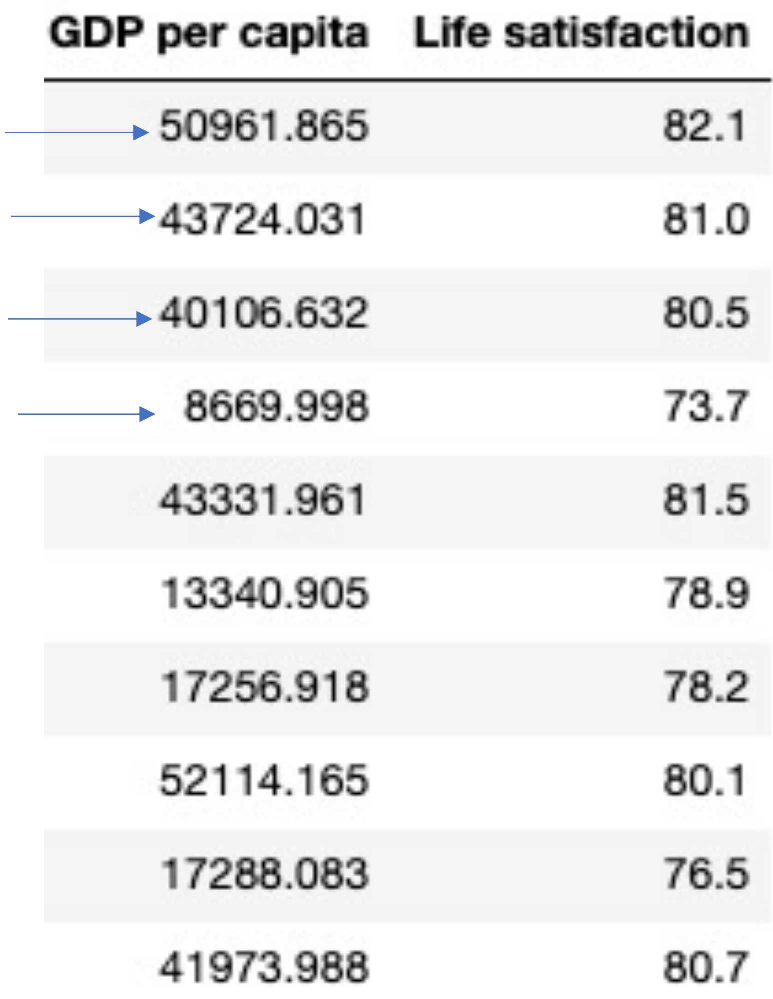


```
import numpy as np
plt.scatter(data['GDP per capita'] , data['Life satisfaction'])
x = np.array([1000 , 100000])
t_0 = 76.61443338
t_1 = 8.82017196e-05
plt.plot(x , t_0 + t_1*x , c = 'red')
plt.xlabel('GDP per capita')
plt.ylabel('Life satisfaction')
plt.show()
```



$$life\ satisfaction = \theta_0 + \theta_1 * \ GDP\ per\ Capita$$

ML (Linear Assumption):



	GDP per capita	Life satisfaction	
$x^{(1)}$ →	50961.865	82.1	$y^{(1)}$
$x^{(2)}$ →	43724.031	81.0	$y^{(2)}$
$x^{(3)}$ →	40106.632	80.5	$y^{(3)}$
$x^{(4)}$ →	8669.998	73.7	$y^{(4)}$
	43331.961	81.5	
	13340.905	78.9	
	17256.918	78.2	
	52114.165	80.1	
	17288.083	76.5	
	41973.988	80.7	

$$\hat{y}^{(i)} = \theta_0 + \theta_1 \times x^{(i)}$$

ML (Example):

Main assumption: The data follows a linear model:

$$life\ satisfaction = \theta_0 + \theta_1 * GDP\ per\ Capita$$

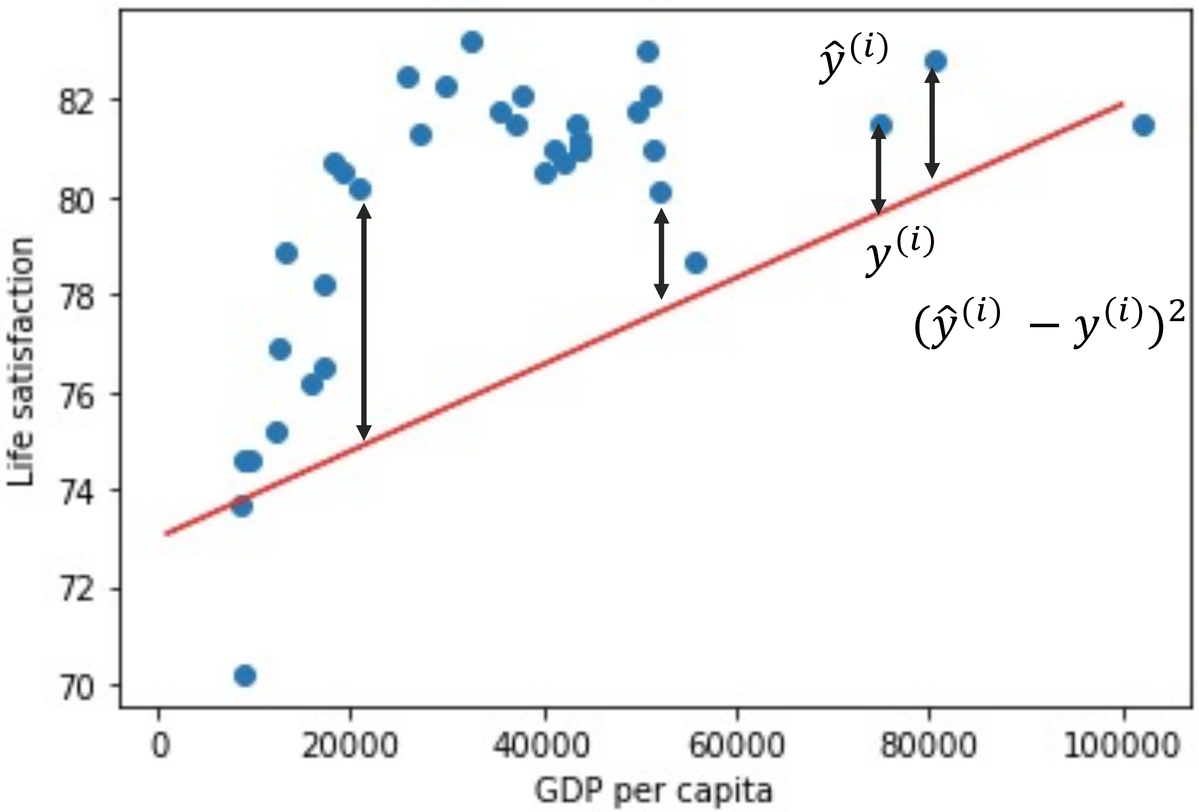
- ➢How you know which values make your model perform best?
    - ○ Fitness Function
    - ○ Cost Function (typically used for linear regression problems.)

Linear Regression algorithm comes into play:

you feed it your training examples and it finds the parameters that make the linear model fit best to your data. This is called *training* the model.

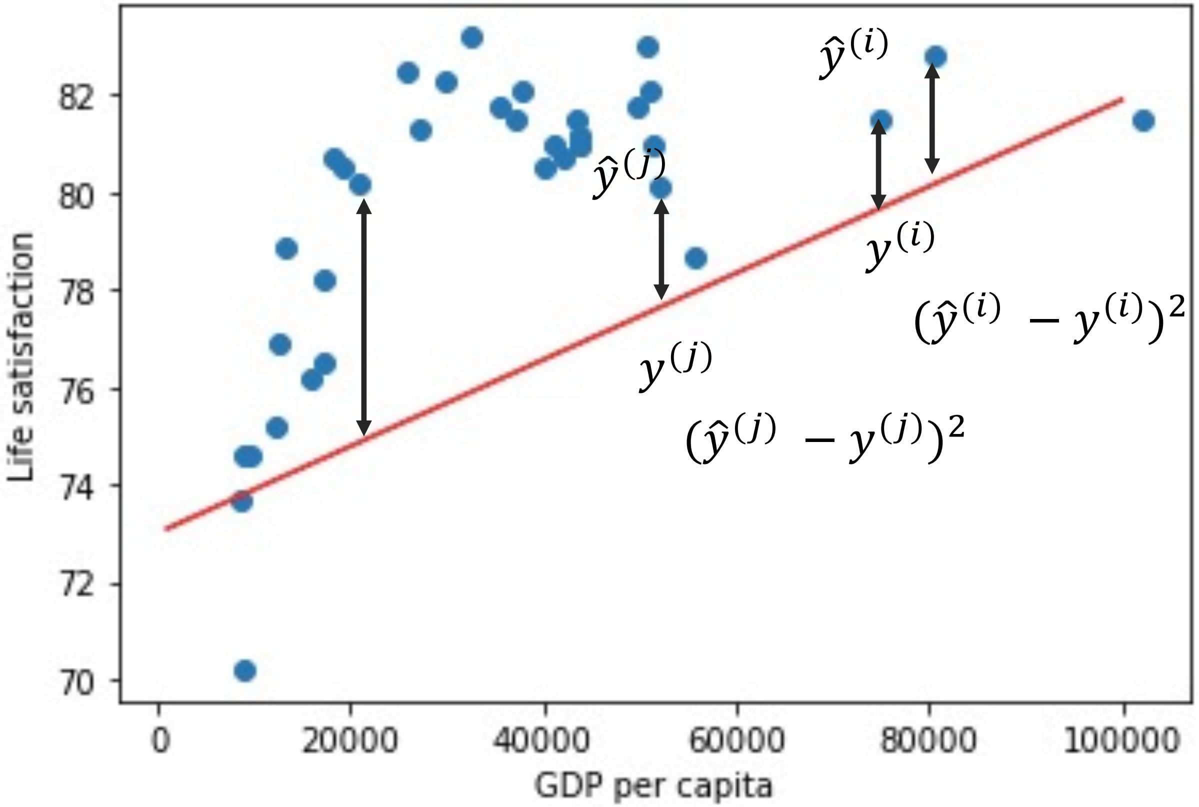
ML (Example):

Cost Function:



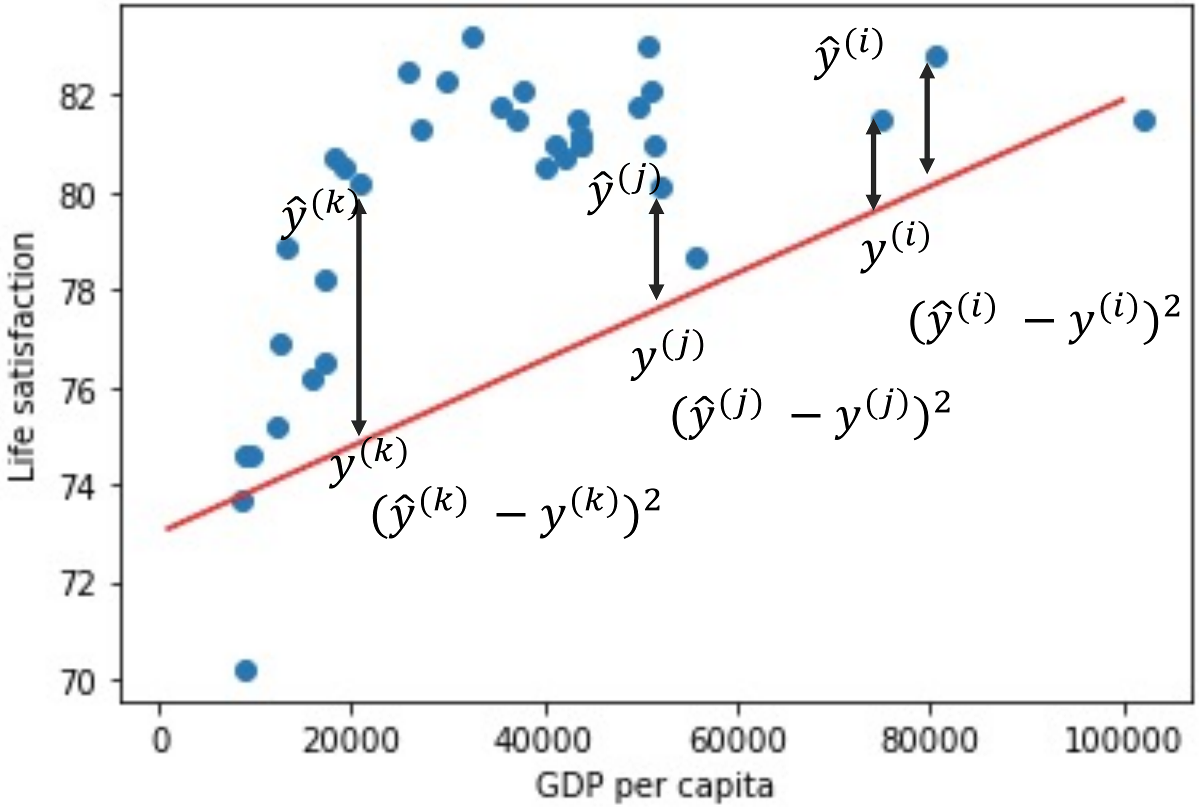
ML (Example):

Cost Function:



ML (Example):

Cost Function: Root Mean Square Error

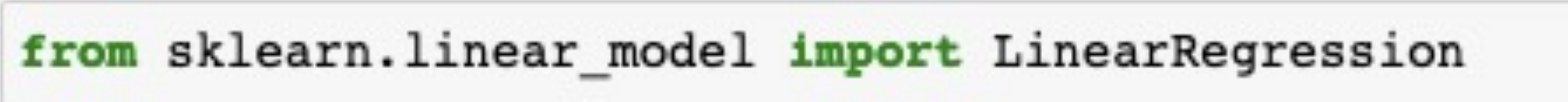


$$\mathrm{RMS}E = \sqrt{\frac{1}{m}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2}$$

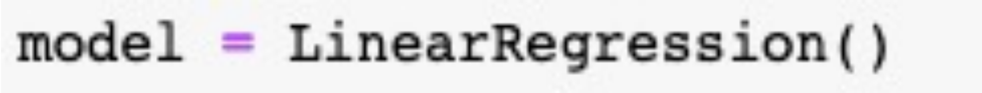
Always positive

ML (Example):

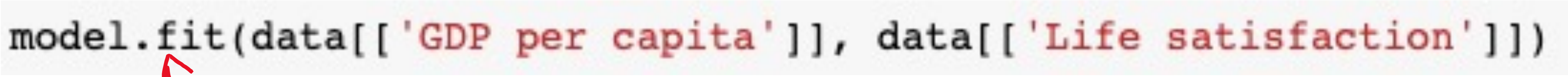
Sklearn (Python library for sklearn)



```
from sklearn.linear_model import LinearRegression
```



```
model = LinearRegression()
```

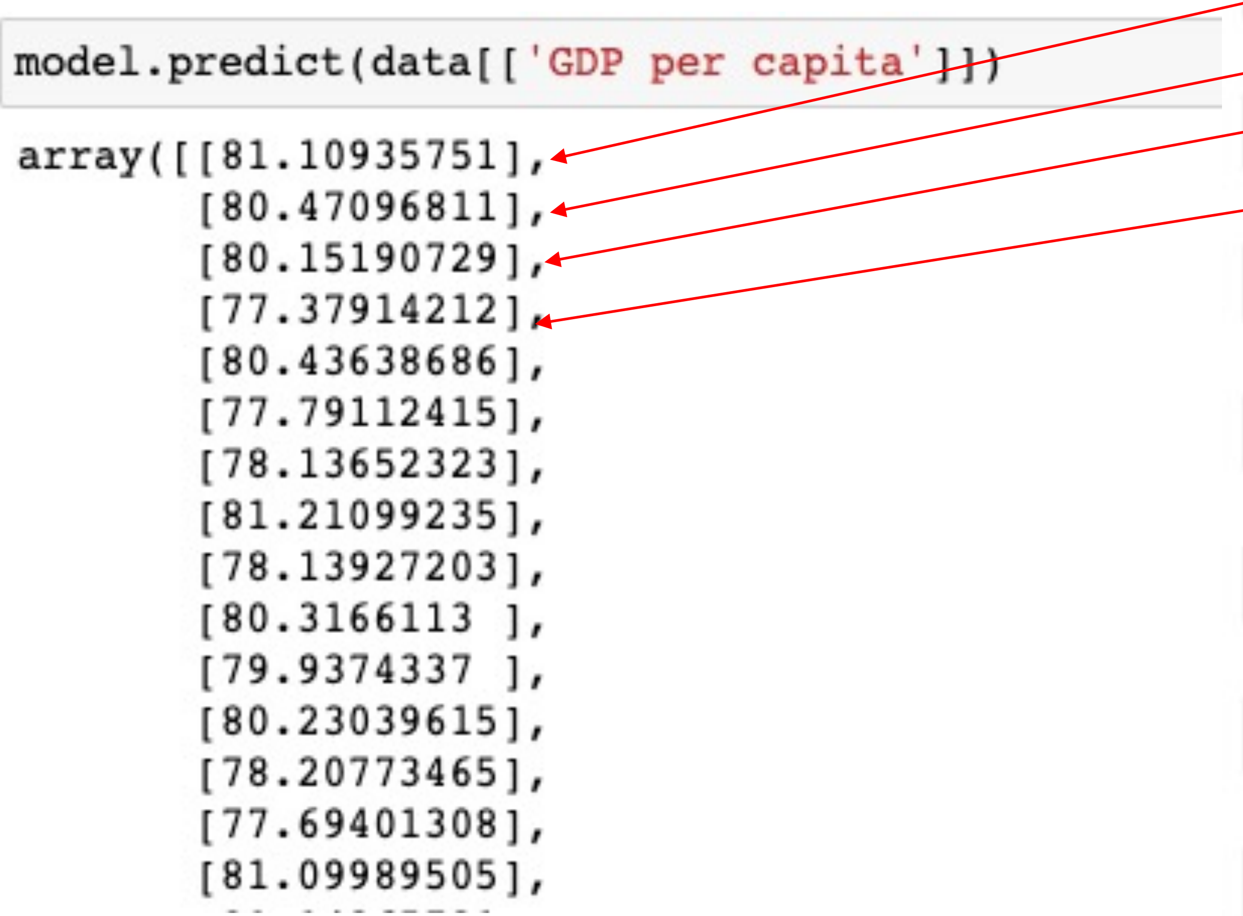


```
model.fit(data[['GDP per capita']], data[['Life satisfaction']])
```

Model Training

ML (Example):

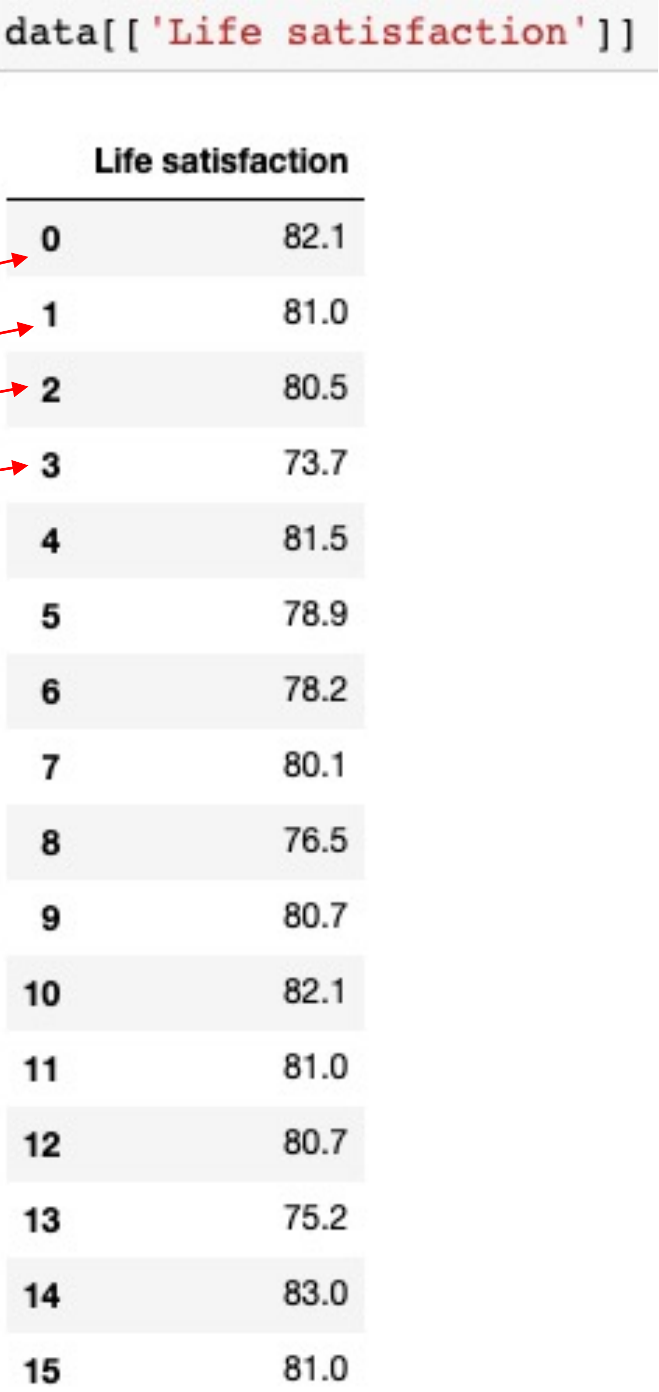
Sklearn (Python library for sklearn)



```
model.predict(data[['GDP per capita']])
```

```
array([[81.10935751],
       [80.47096811],
       [80.15190729],
       [77.37914212],
       [80.43638686],
       [77.79112415],
       [78.13652323],
       [81.21099235],
       [78.13927203],
       [80.3166113 ],
       [79.9374337 ],
       [80.23039615],
       [78.20773465],
       [77.69401308],
       [81.09989505],
```

Model Prediction



```
data[['Life satisfaction']]
```

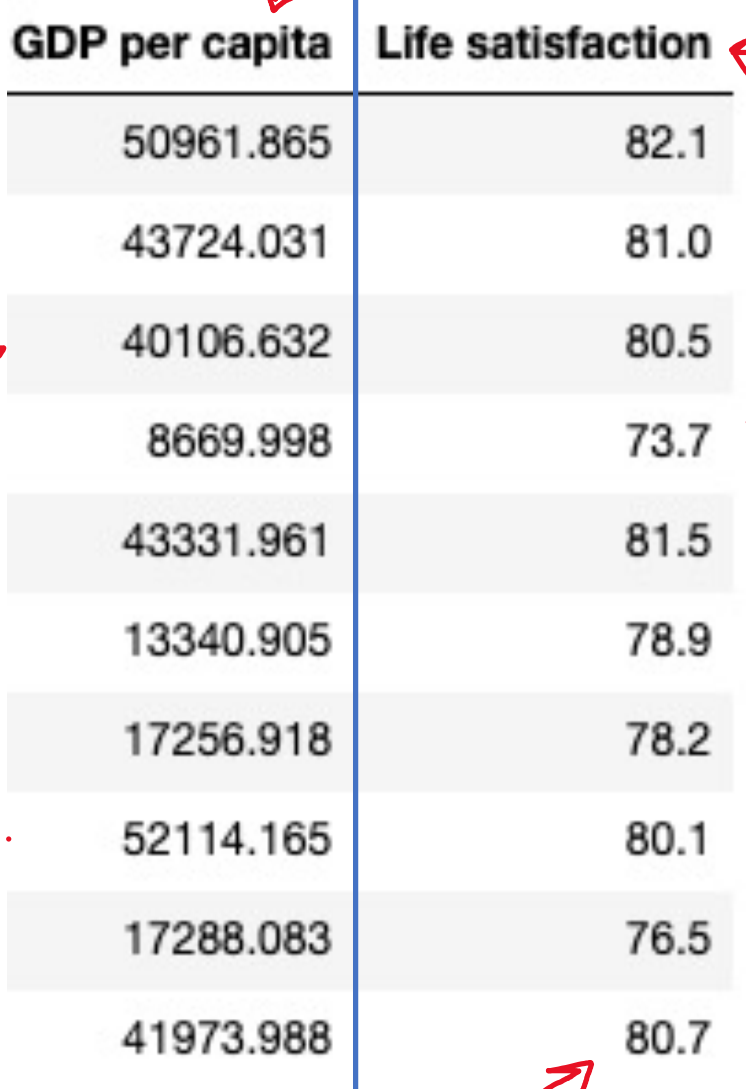
	Life satisfaction
0	82.1
1	81.0
2	80.5
3	73.7
4	81.5
5	78.9
6	78.2
7	80.1
8	76.5
9	80.7
10	82.1
11	81.0
12	80.7
13	75.2
14	83.0
15	81.0

ML (Example):

Predictor (A combination of attributes)

Target variable

Samples

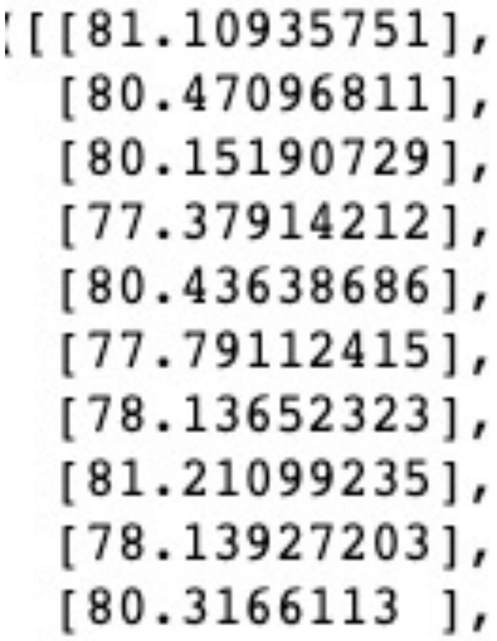


GDP per capita	Life satisfaction
50961.865	82.1
43724.031	81.0
40106.632	80.5
8669.998	73.7
43331.961	81.5
13340.905	78.9
17256.918	78.2
52114.165	80.1
17288.083	76.5
41973.988	80.7

Assumption of Linearity

Training (Minimizing RMSE)

Predictions



```
[[81.10935751],
 [80.47096811],
 [80.15190729],
 [77.37914212],
 [80.43638686],
 [77.79112415],
 [78.13652323],
 [81.21099235],
 [78.13927203],
 [80.3166113 ],
```

ML :

Note: In general their might be more than one attribute:

- In this case, the first attribute of simple (i) is represented by variable $x_1^{(i)}$,
- The second attribute would be $x_2^{(i)}$
- ....
- The attribute p would be $x_p^{(i)}$

The linear assumption: $$\hat{y}^{(i)} = \underbrace{\theta_0 + \theta_1 \times x_1^{(i)} + \theta_2 \times x_2^{(i)} + \ldots + \theta_p \times x_p^{(i)}}_{\text{Hyperplane}}$$

Note: in some texts instead of $\hat{y}^{(i)}$, they use $h_\theta(x^i)$

ML (Example):

How the training part works? (The minimization of RMSE)

Min $\quad \mathrm{RM}SE = \sqrt{\frac{1}{m}\sum_{i=1}^{n}(\hat{y}^{(i)} \ - y^{(i)})^2}$

Is equal to Min $\quad (\theta^T X \ - y)^2$

$$\theta = [\theta_0 \quad \theta_1 \quad \theta_2 \ \ldots \ \theta_p]$$

$$X = \begin{pmatrix} 1 & x_1^1 & x_2^1 & \cdots & x_p^1 \\ 1 & x_1^2 & x_2^2 & \cdots & x_p^2 \\ \vdots & \vdots & \cdots & \vdots & \\ 1 & x_1^n & x_2^n & \cdots & x_p^n \end{pmatrix}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

ML (Example):

How the training part works? (The minimization of RMSE)

Min $(\theta^T X - y)^2$

$$\arg min_{\theta \in \mathbb{R}^{p+1}} (\theta X - Y)^T (\theta X - Y)$$

$$\nabla_\theta (\theta X - Y)^T (\theta X - Y) = 0$$

$$-2 X^T(y - \theta X) = 0$$

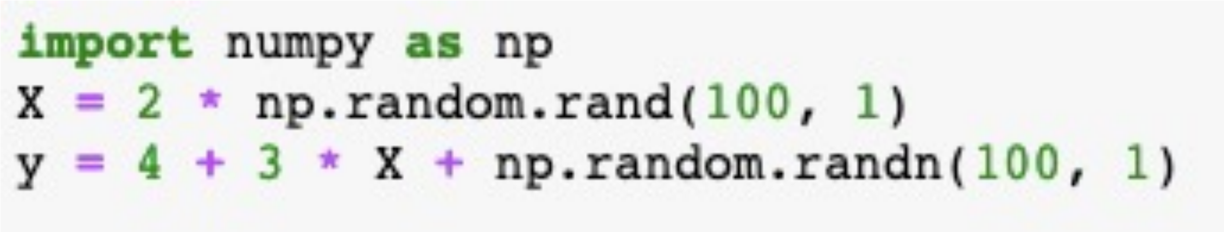
Normal Equation

$$\theta = (X^T X)^{-1} X^T y$$

It has a solution only when $(X^T X)^{-1}$ is inversible (when it's determinant is non-zero).

ML (Example):

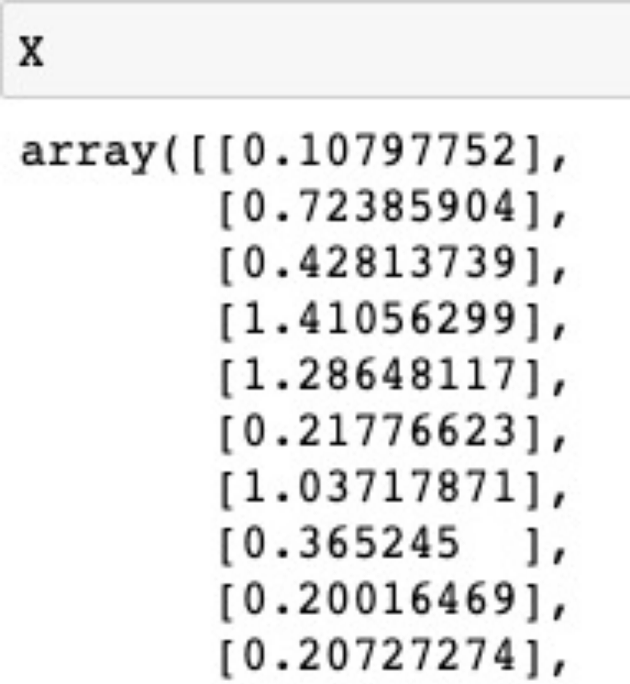
Let’s test the normal equation by generating random data that follow linear pattern:



```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

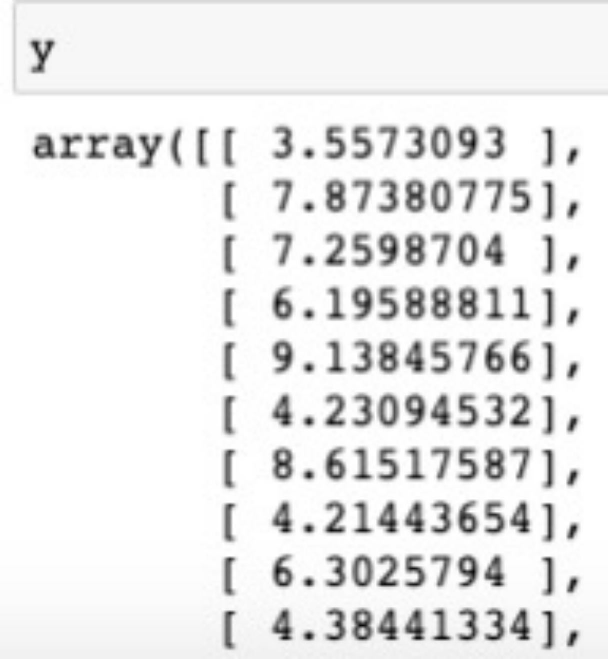
Predictor

Target Variable



```
X
```

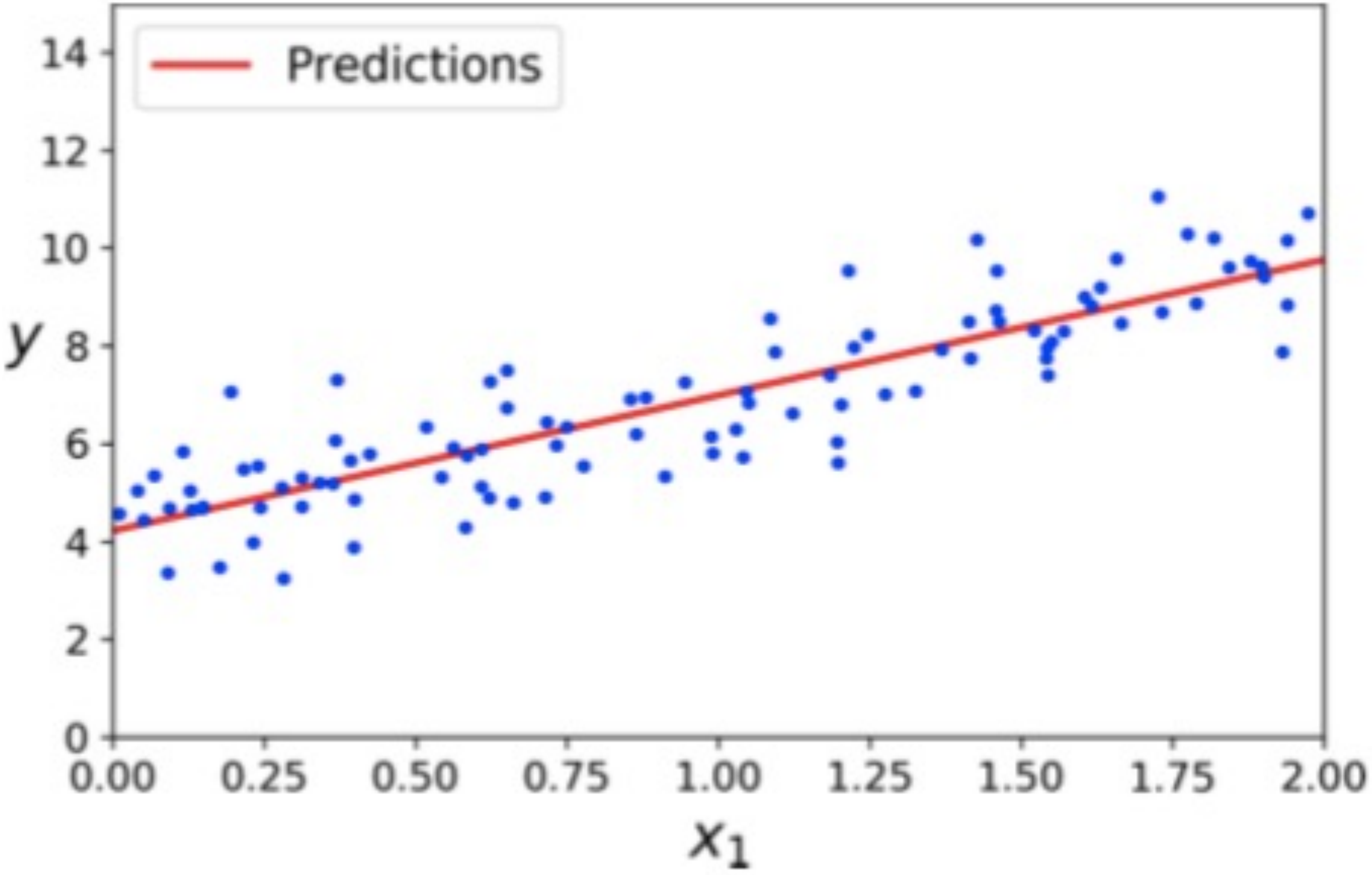
```
array([[0.10797752],
       [0.72385904],
       [0.42813739],
       [1.41056299],
       [1.28648117],
       [0.21776623],
       [1.03717871],
       [0.365245  ],
       [0.20016469],
       [0.20727274],
```



```
y
```

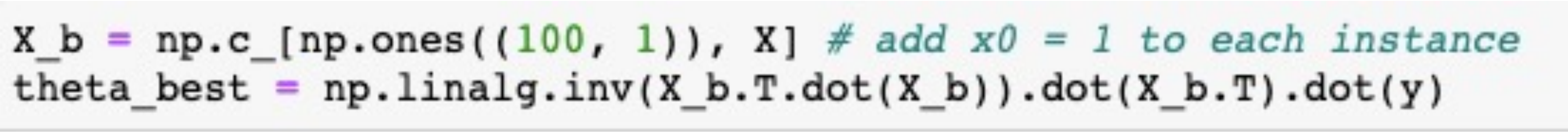
```
array([[ 3.5573093 ],
       [ 7.87380775],
       [ 7.2598704 ],
       [ 6.19588811],
       [ 9.13845766],
       [ 4.23094532],
       [ 8.61517587],
       [ 4.21443654],
       [ 6.3025794 ],
       [ 4.38441334],
```

ML (Example):



Predictions
14
12
10
8
6
4
2
0
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x_1

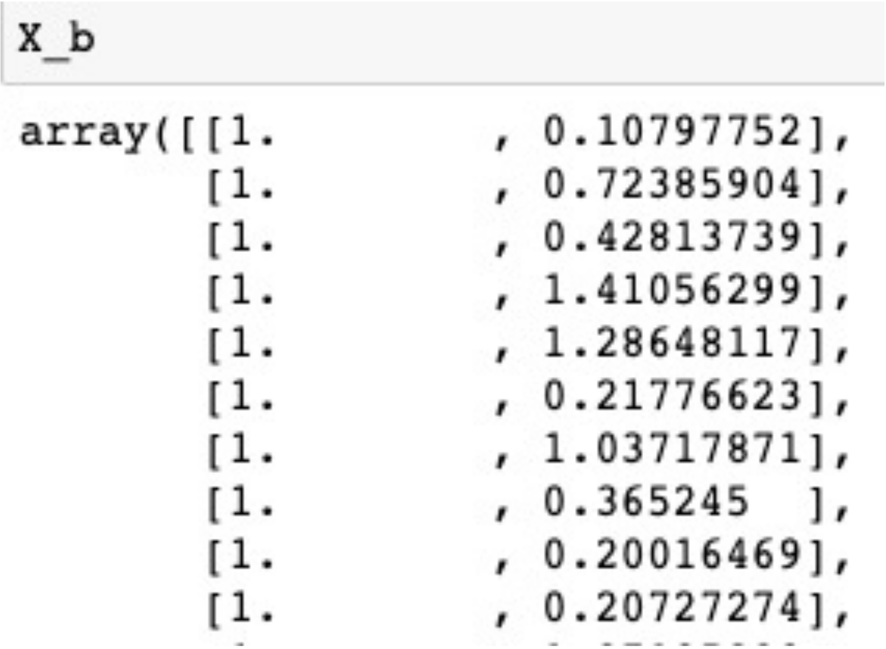
ML (Example):



```
X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

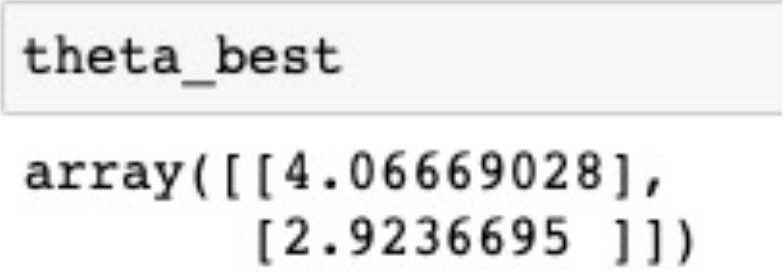
Create matrix X

Predictor



```
X_b
```

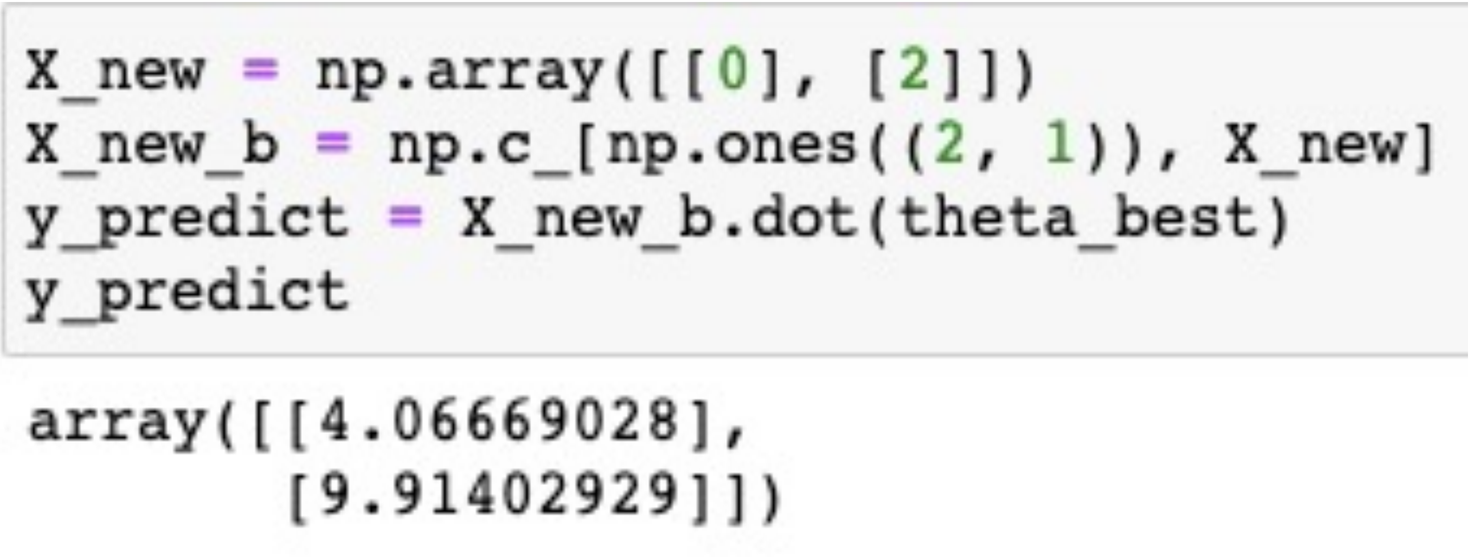
```
array([[1.        , 0.10797752],
       [1.        , 0.72385904],
       [1.        , 0.42813739],
       [1.        , 1.41056299],
       [1.        , 1.28648117],
       [1.        , 0.21776623],
       [1.        , 1.03717871],
       [1.        , 0.365245  ],
       [1.        , 0.20016469],
       [1.        , 0.20727274],
```



```
theta_best
```

```
array([[4.06669028],
       [2.9236695 ]])
```

ML (Example):



```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
y_predict
```

```
array([[4.06669028],
       [9.91402929]])
```

ML (Exercise):

Calculate normal equation for the dataset of GDP per capita / Life satisfaction.

ML (Gradient Descent):

- Problems with normal equations:

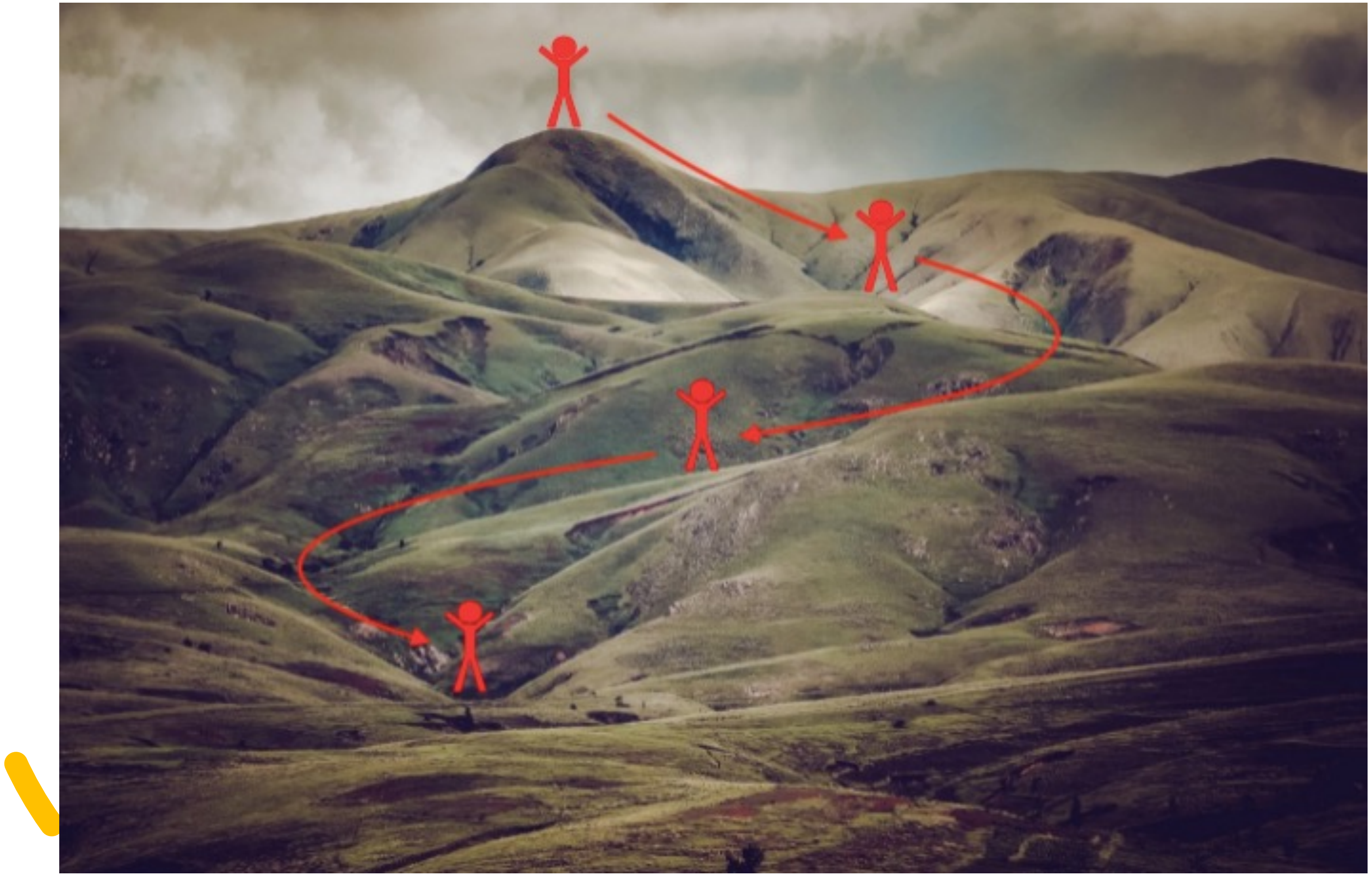
1. In many real cases $(X^T X)^{-1}$ is not invertible,
2. Even if it is for bid data sets the computational cost is $O(n^3)$ or $O(n^{2.4})$.

- So, instead of calculating θ from the normal equation, the learning algorithms use a technique to estimate this value which is called Gradient Descent.

1. Start with some initial parameters θ,

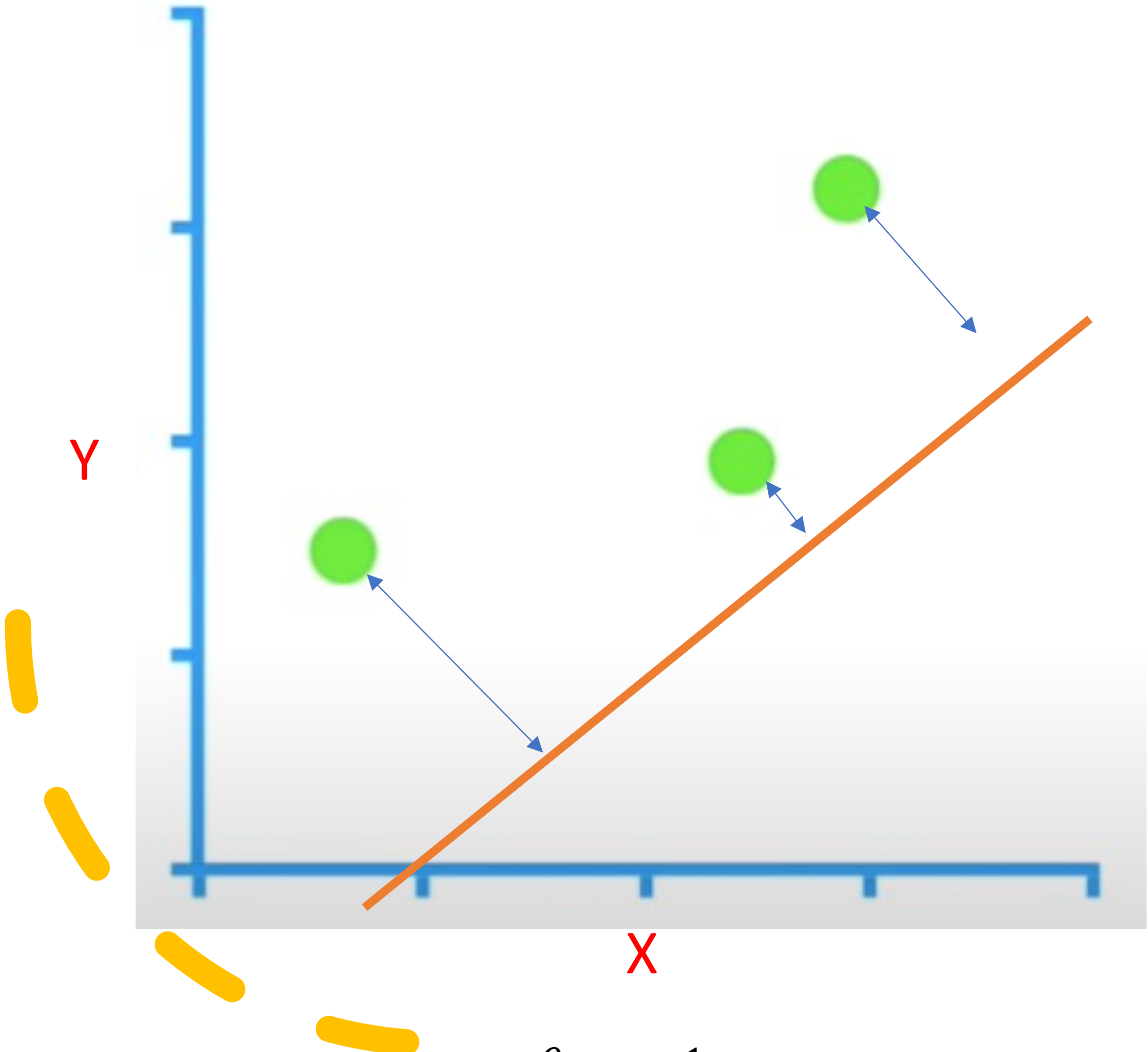
2. Tweaking the parameters (θ) iteratively, in a way that it reduces the cost function.

ML (Gradient Descent-Example):

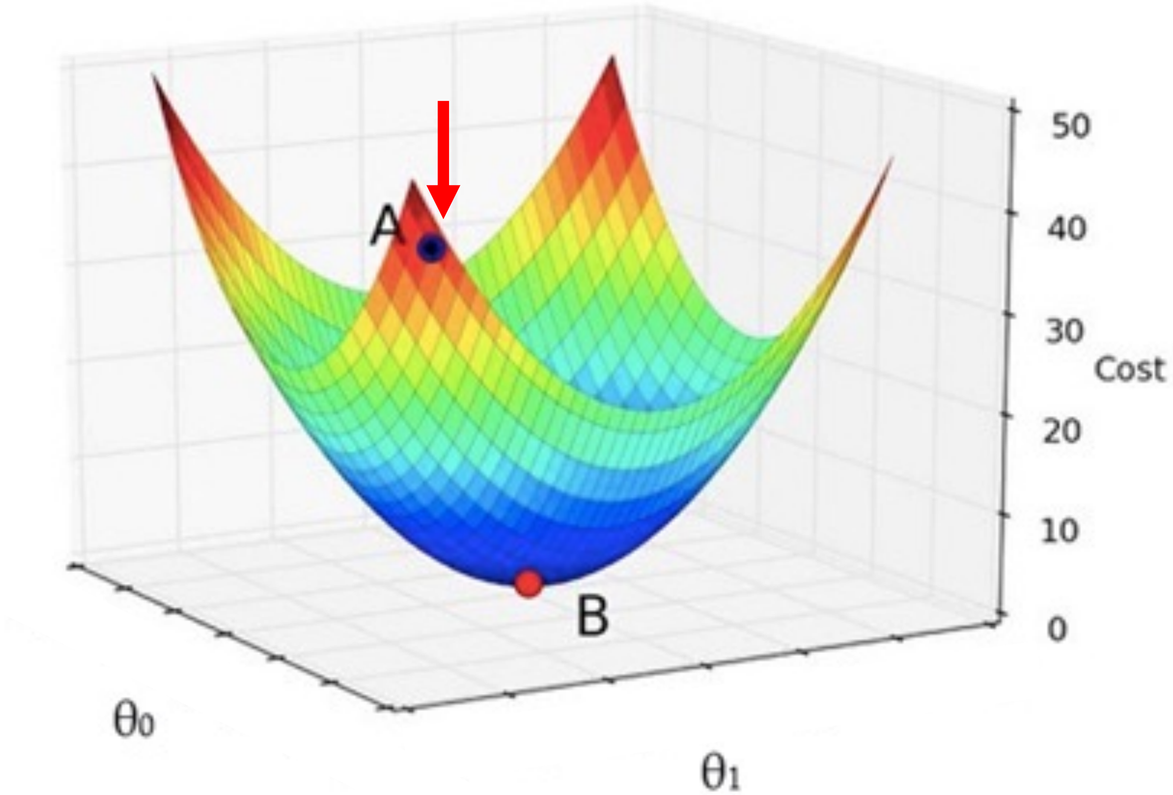


1. Start from a position (x, y),
2. Find the negative slope (to descend)
3. Take appropriate size step.

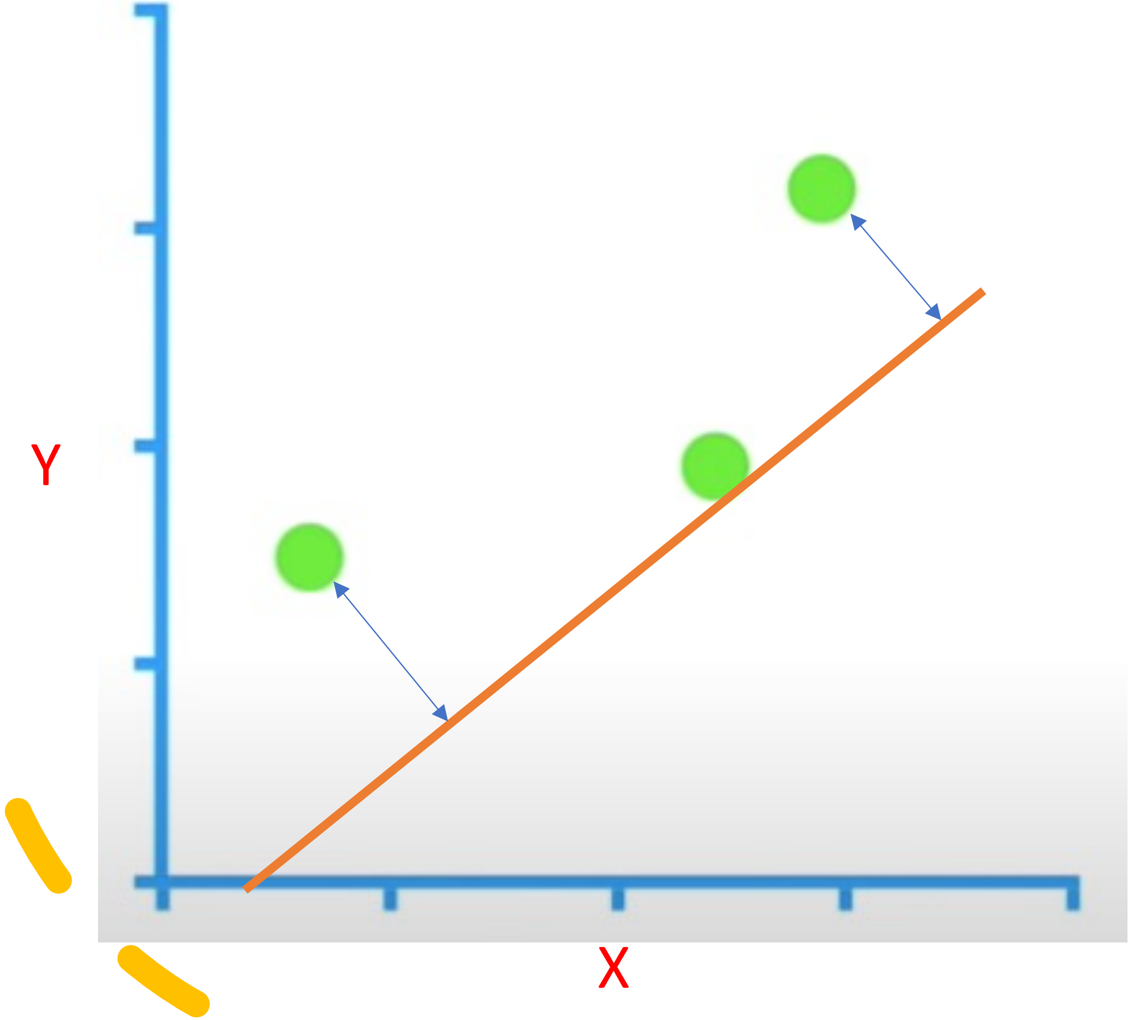
ML (Gradient Descent-Example):



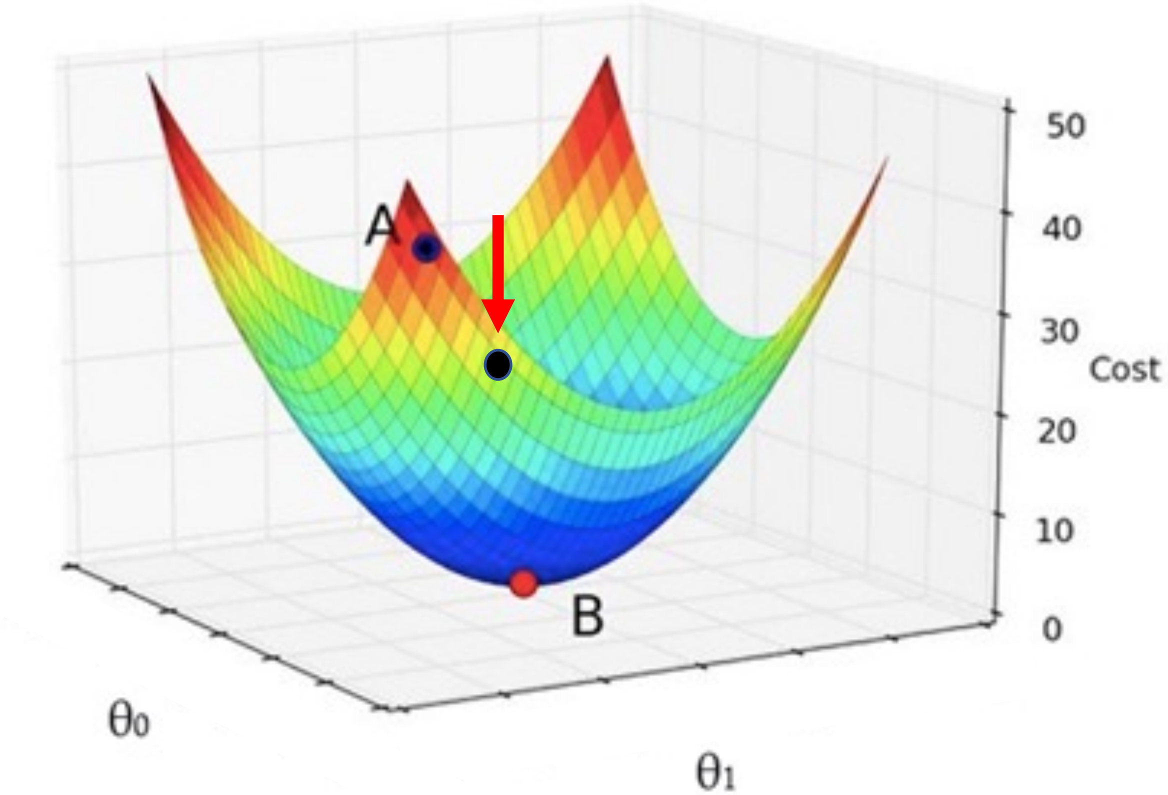
$\theta_0 = -1$
$\theta_1 = 1$



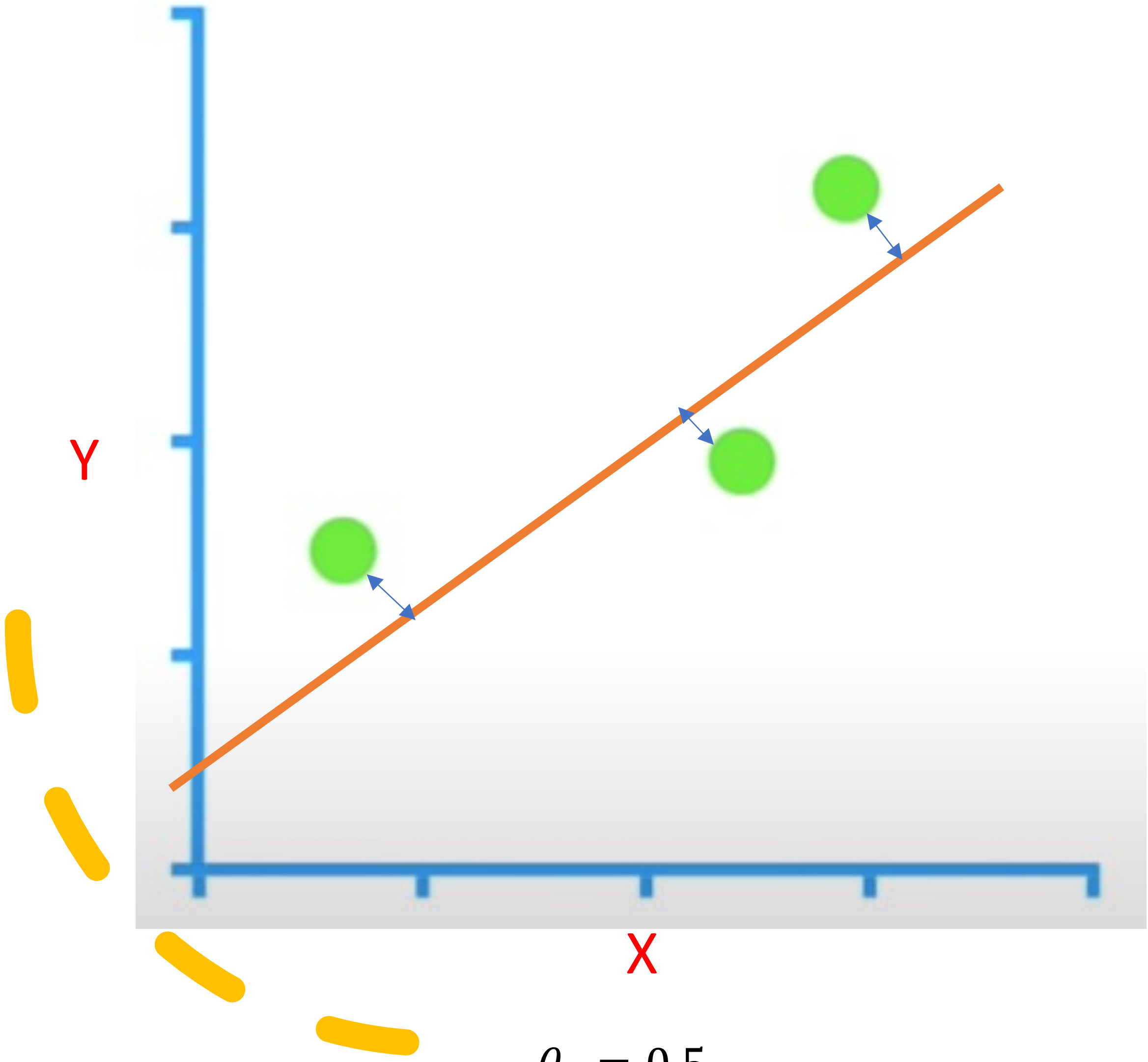
ML (Gradient Descent-Example):

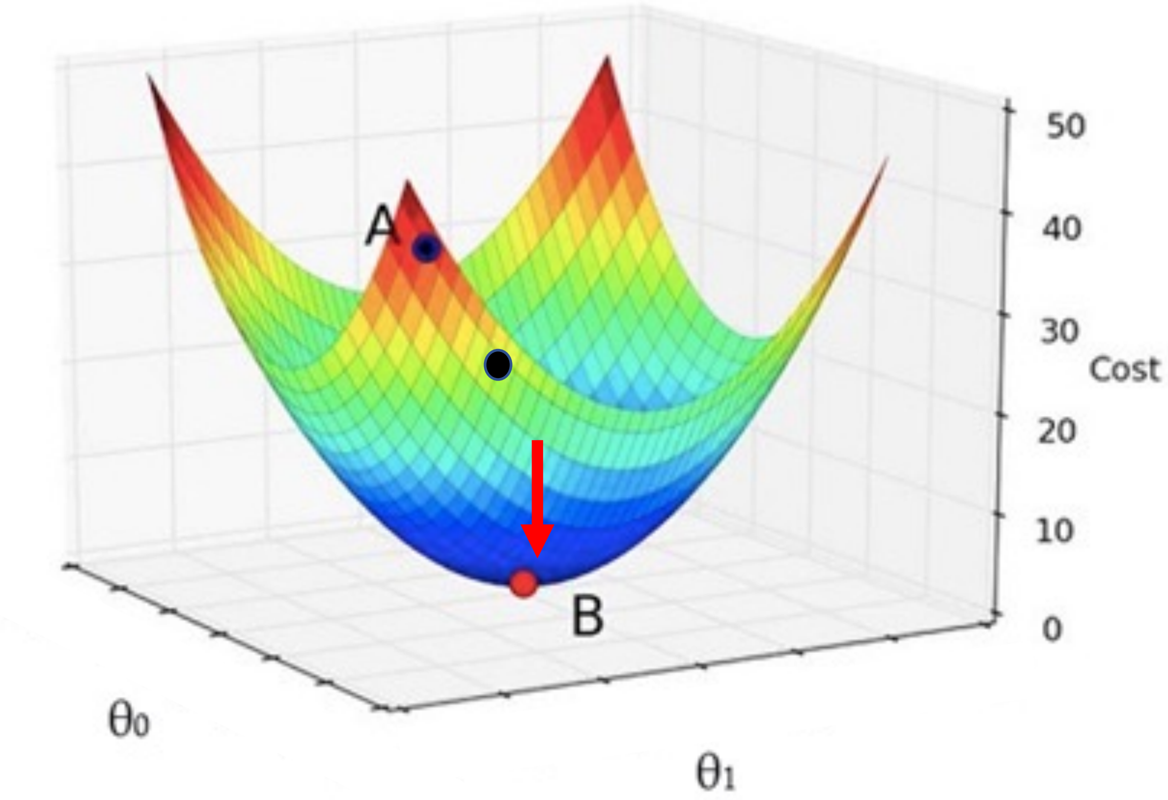


$\theta_0 = -0.5$
$\theta_1 = 1$



ML (Gradient Descent-Example):





$\theta_0 = 0.5$

$\theta_1 = 0.6$

ML (Gradient Descent-Example):

- Suppose we start with initial θ_0 and θ_1 .
- In which direction we should go to reduce the cost function?

$$MSE = \underbrace{(h_\theta\left(x^{(1)}\right) - y^{(1)})^2}_{} + \underbrace{(h_\theta\left(x^{(2)}\right) - y^{(2)})^2}_{} + \underbrace{(h_\theta\left(x^{(3)}\right) - y^{(3)})^2}_{}$$

$$\frac{d(MSE)}{d(\theta_0)} = \frac{d(MSE)}{d(h_\theta)} \times \frac{d(h_\theta)}{d(\theta_0)}$$

$$\frac{d(MSE)}{d(\theta_0)} = \overbrace{2\left(h_\theta\left(x^{(1)}\right) - y^{(1)}\right) \times 1}^{} + \overbrace{2\left(h_\theta\left(x^{(2)}\right) - y^{(2)}\right) \times 1}^{} + \overbrace{2\left(h_\theta\left(x^{(3)}\right) - y^{(3)}\right) \times 1}^{}$$

$$\frac{d(MSE)}{d(\theta_1)} = \frac{d(MSE)}{d(h_\theta)} \times \frac{d(h_\theta)}{d(\theta_1)}$$

$$\frac{d(MSE)}{d(\theta_1)} = 2\left(h_\theta\left(x^{(1)}\right) - y^{(1)}\right) \times x^{(1)} + 2\left(h_\theta\left(x^{(2)}\right) - y^{(2)}\right) \times x^{(1)} + 2\left(h_\theta\left(x^{(3)}\right) - y^{(3)}\right) \times x^{(1)}$$

ML (Gradient Descent-Example):

- Step sizes:

Step size (θ_0) = $\overset{\frac{d(MSE)}{d(\theta_0)}}{\overset{\downarrow}{\text{slope}}} \times \mathit{learning\ rate}$

Step size (θ_1) = $\underset{\frac{d(MSE)}{d(\theta_1)}}{\underset{\uparrow}{\text{slope}}} \times \mathit{learning\ rate}$

Update Functions:

new θ_0 = previous θ_0 - step size θ_0
new θ_1 = previous θ_1 - step size θ_1

ML (Gradient Descent-summary):

1. Random initialization of parameters ($\theta_0, \theta_1, \theta_2$,...),
2. Calculate the slopes using gradient descent and chain rule,
3. Compute the steps using a pre-defined learning rate,
4. Update the parameters,

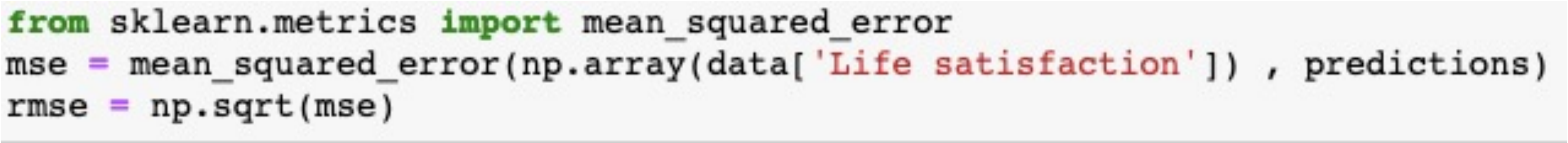
Note: In Machine learning, all the parameters that should be pre-defined in order to use the model are called hyper parameters. Hyper parameters are different from parameters. The parameters will be learned during training,...

ML (Error Calculation):

Calculate the Root Mean Square Error for the predictions you made using linear regression for the dataframe GDP per capita/ life satisfaction(Use the fonctionalities of numpy).

ML (Error Calculation):

Sub-module metrics and the function mean_squared_error of Sklearn.



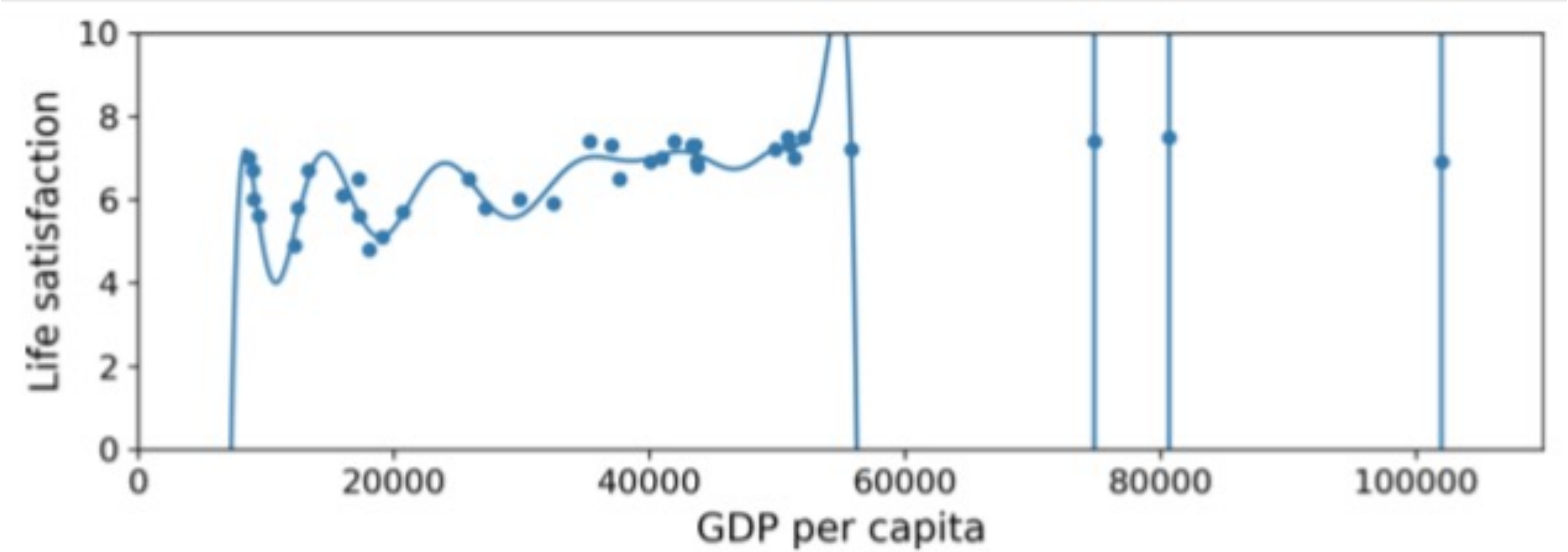
```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(np.array(data['Life satisfaction']) , predictions)
rmse = np.sqrt(mse)
```

ML (Question):

- Is this error reliable for future predictions?
- Does it mean that our model will perform the best to predict?

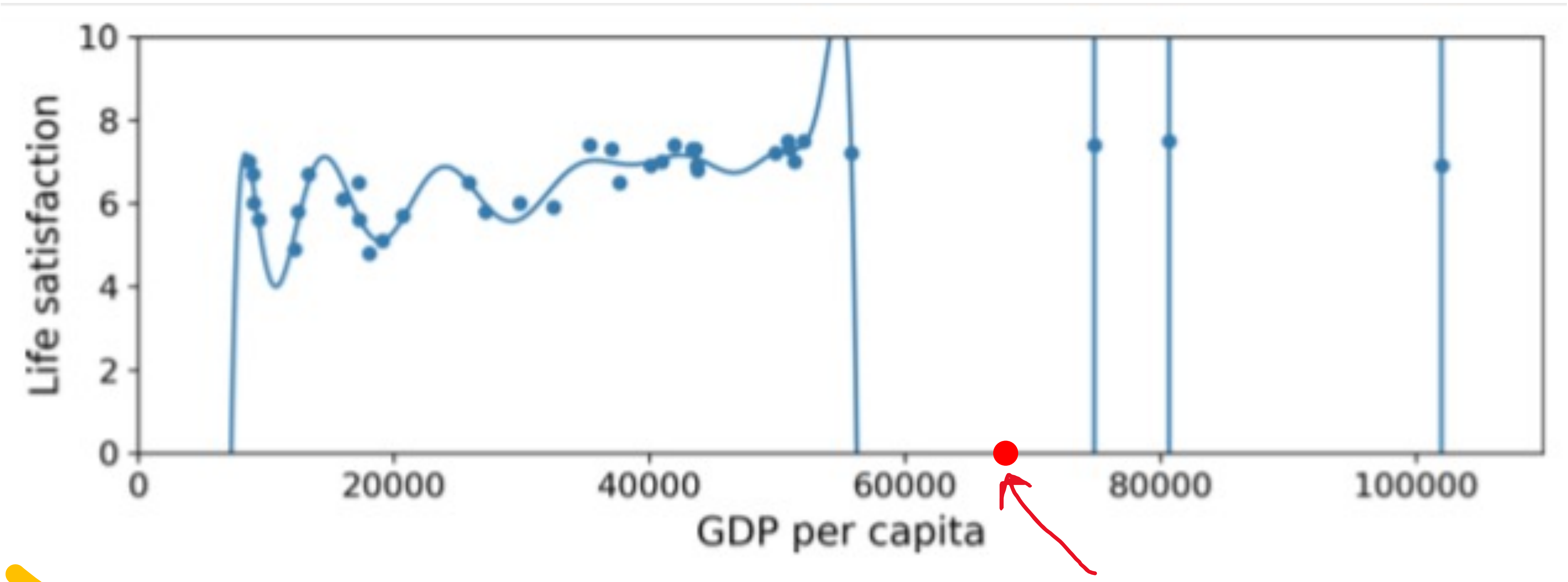
ML (Challenges of training):

Overfitting



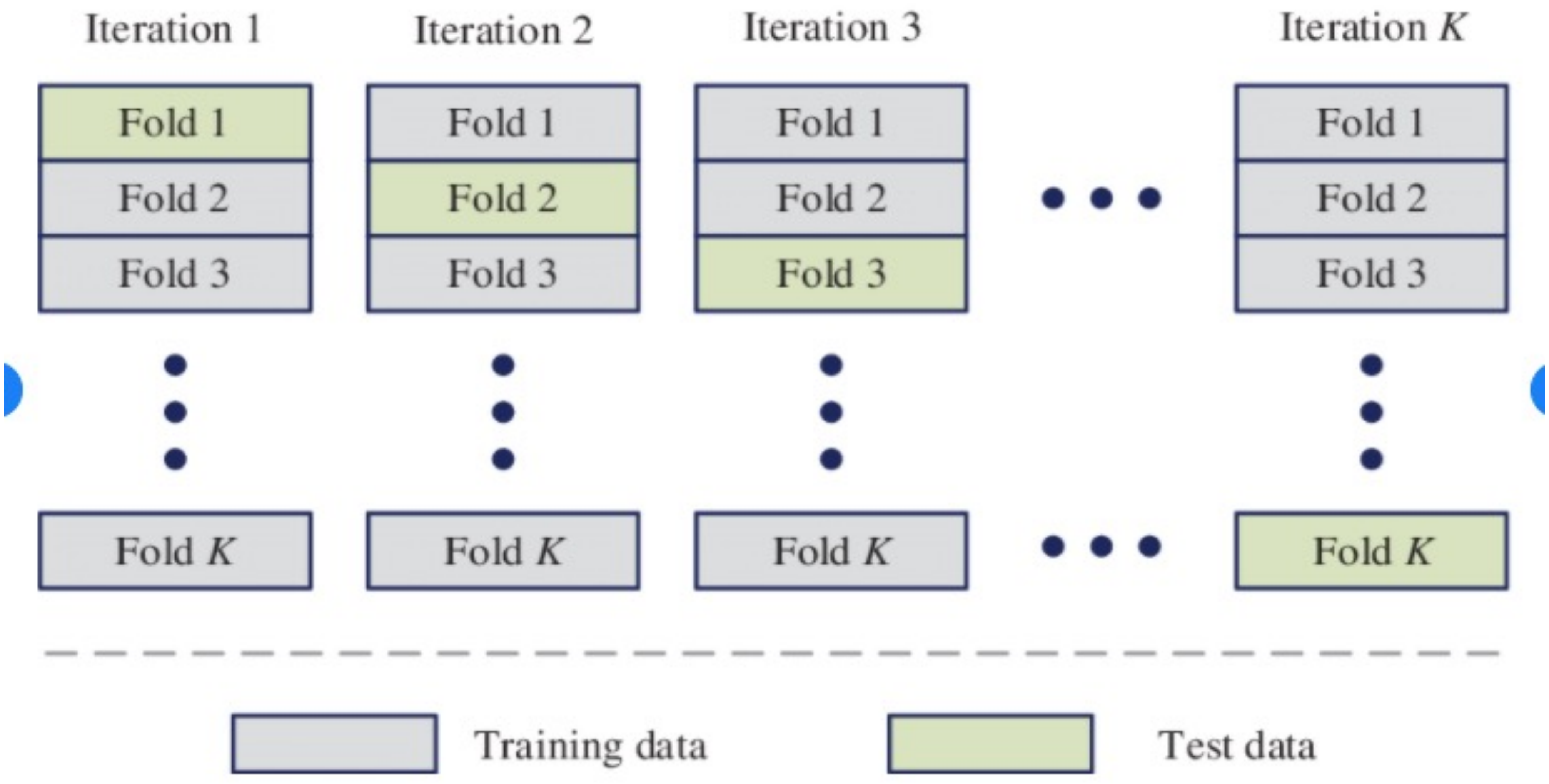
ML (Challenges of training):

Overfitting



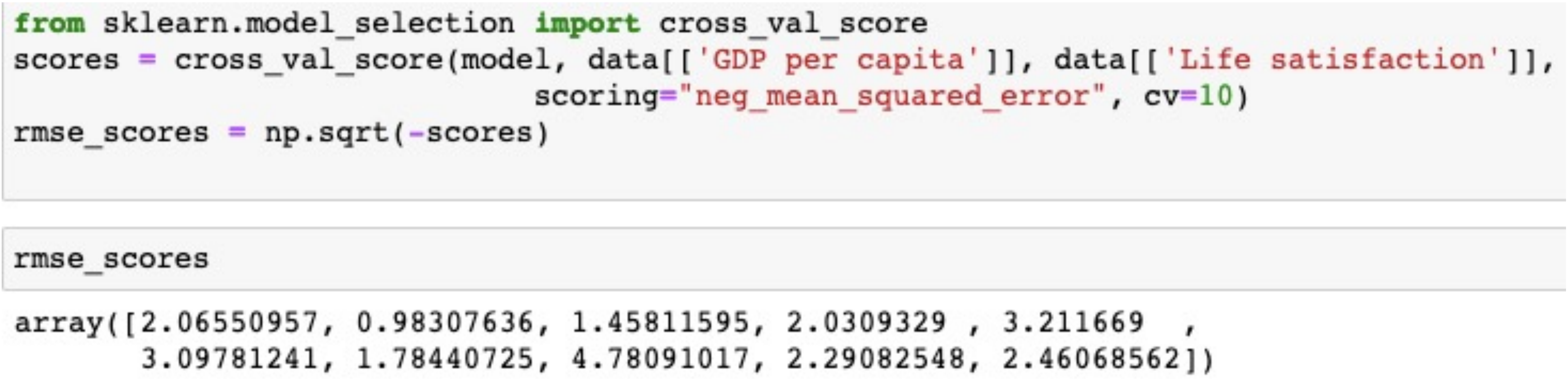
What would be the prediction value for this point?

ML (Concept of cross validation):



K-fold cross-validation method.

ML (Concept of cross validation):



```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, data[['GDP per capita']], data[['Life satisfaction']],
                                 scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)
```

```
rmse_scores
```

```
array([2.06550957, 0.98307636, 1.45811595, 2.0309329 , 3.211669  ,
       3.09781241, 1.78440725, 4.78091017, 2.29082548, 2.46068562])
```