For this session, you will need to install Python version 3.6. You will also need the following libraries: `numpy, tkinter, random` and `time`. Random and Time are already included in the Python default installation. Numpy need to be installed manually. It's very easy, firstly, open a terminal and type:

```
pip install numpy
```

That's it. Let's get to business.

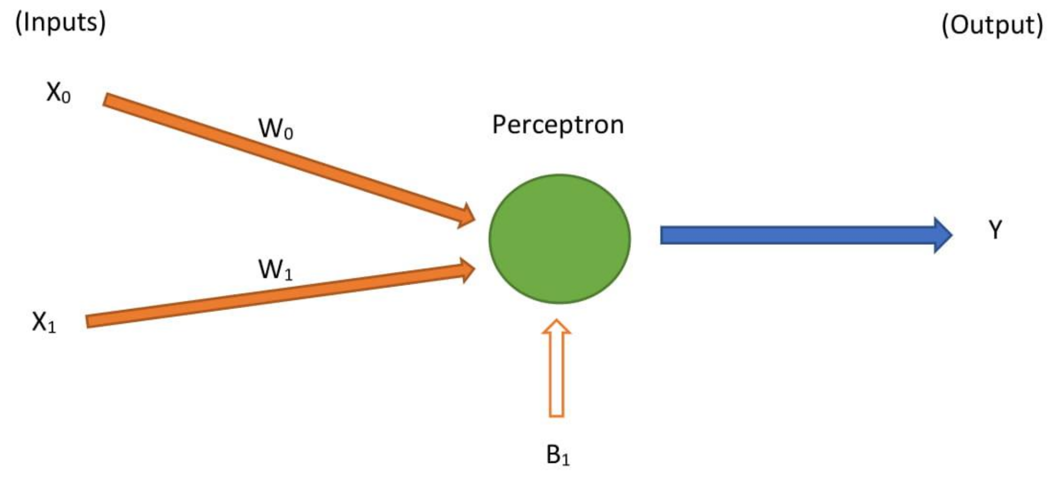
Neuron: Something that receives inputs, do some magic with its algorithm, then gives an output.

Perceptron: Represent a neuron. It Receives X inputs, do some magic with its algorithm, then returns 1 output.

Each **connection** (the orange arrows in the diagram below) that the output is redirected to a Perceptron is Weighted.

A **Weight** is a number that if the Perceptron guesses the output incorrectly, we will tweak the Weight in order to make the Perceptron more accurate. Weights here are represented with the letter W.

Finally, we will need another value which is a bit special. This is called a **Bias** (Represented by B here). A bias is here to make sure the Perceptron learn in the right direction (Think of the linear function "y=ax+b", where the Bias is "b"). A bias is, in itself, a weight but only for the Perceptron. We will talk about it later on.



Here is what will happen in our Perceptron:

1. The main algorithm will calculate the result based on the Weights, the Inputs and the Bias. This is very straightforward: It's the sum of every input multiplied by its weight, including the Bias. This is represented by a **Matrices Dot Product**, which is basically a multiplication between two matrices. It would look like that in our case, with V representing the result of this Matrices Dot Product:

$$V = \begin{bmatrix} W_0 & W_1 \end{bmatrix} * \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} + B_1$$

   The mathematical operation looks like that:

$$V = (W_0 * X_0) + (W_1 * X_1) + B_1$$

2. Once we got V, we need to make sure to normalize it between two known value, because the Perceptron is a binary classifier, meaning it can only work if there are 2 possible output. This is called an **Activation Function**. Here, we can use the **Sign function**, which classify any value to either -1 or 1:

$$W = sgn(V)$$

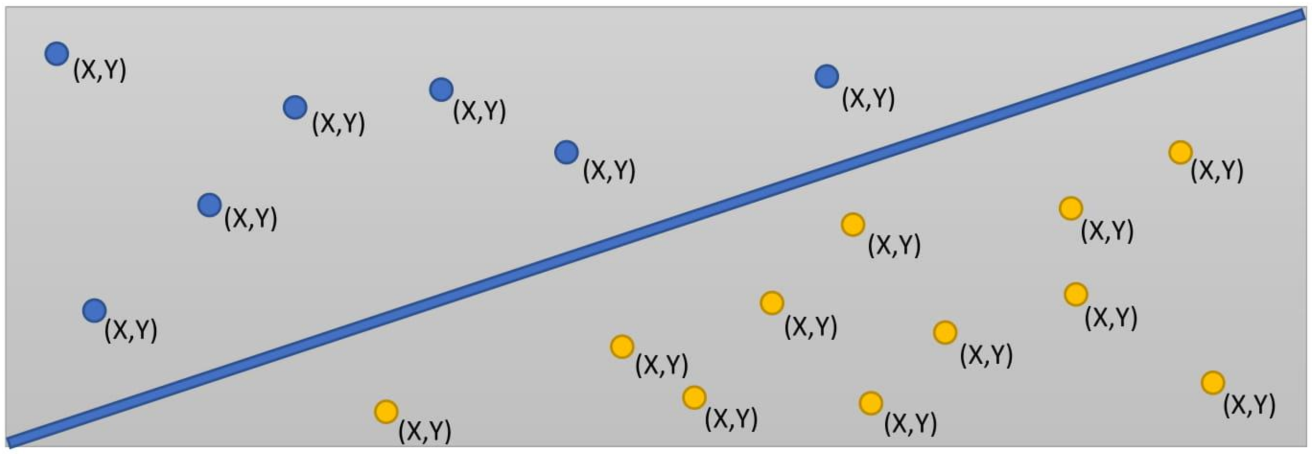
   $\underline{W}$ is the Sign function result
   $\underline{V}$ is our previous value.

That's it!

Here is a simple case scenario where we can use a Perceptron.

Let's suppose we have a canvas with a line splitting it in 2. Inside the canvas, we have points, that are either one side of the line or the other. What our Perceptron will need to do is classify these points into either they are on one side or the other, based on their position (X,Y). If the point is above the line, our Perceptron should return 1. If it's below the line, our Perceptron should return a value of -1. If its on the line, it should return 0.

Here is the canvas:



The technique we will use to **Train** our Perceptron is known as **Supervised Learning**. This simply mean that we will **Feed Foward** (give inputs) to our Perceptron of an answer we already know.

For example, let's say we Feed Forward the inputs (10, 5) and the answer is 1. If the Perceptron guesses its 1, we will say "good job" and continue **Training** the Perceptron with inputs and answers, until it guesses wrong.

Once it guesses wrong, we will tweak the Weights to make him closer from being perfect. This process we will use is called **Gradient Descent**, which we will cover later on. We will do this process until we are satisfied with our Perceptron performance.

But how do we calculate the value to add to the weights, and the bias? Here are the mathematical equations which we will decorticate:

$$Error = ExpectedAnswer - Guess$$

$$Bias = Bias + \ Error * LearningRate$$

And for each input and weight we will do that:

$$\Delta W_n = \ Error * X_n$$

$$W_n = W_n + \ \Delta W_n * LearningRate$$

Guess is the Perceptron guessed answer.

ExpectedAnswer is the real answer that we know.

Bias is the bias

ΔW_n is the value which we need to add to the previous weight value.

W_n is the weight in question.

LearningRate is a value between 0 and 1. The bigger the value, the faster our Perceptron will train. However, too large can mean that our Perceptron will learn incorrectly. A typical **Learning Rate** for this example could be 0.01.

Let’s say re-take our example above with (10,5) as inputs and 1 as the expected answer. Let’s say the Perceptron guessed -1.

Let’s also say our weights are all equals to 0.1 at the moment.

The equations considering the inputs 1 and 2 would then look like that:

$\underline{X_1} = 10$, $\underline{W_1} = 0.1$, $\underline{X_2} = 5$, $\underline{W_2} = 0.1$, $\underline{LearningRate} = 0.01$, $\underline{B_1} = 0.1$, $\underline{ExpectedAnswer} = 1$, $\underline{Guess} = -1$

$$Error = 1 - -1 = 2$$

$$\Delta W_1 = 2 * 10 = 20$$

$$Bias = 2 * 0.01 = 0.02$$

$$W_1 = 0.1 + 20 * 0.01 = 0.3$$

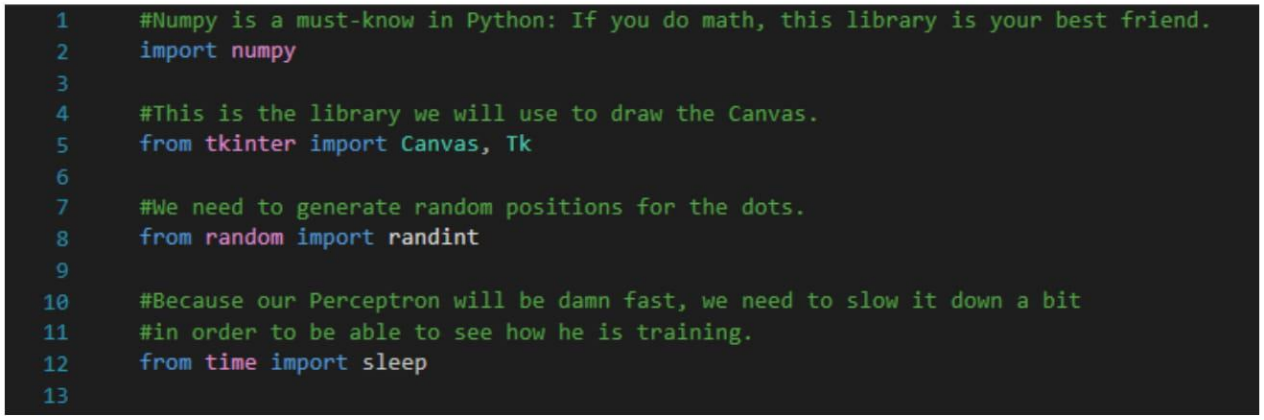
$$\Delta W_2 = 2 * 5 = 10$$

$$W_2 = 0.1 + 10 * 0.01 = 0.2$$

We can now repeat that process over and over again until our Perceptron is perfect. Lets now implement that example in real life. We will do it in Python, in the 3.6 version, since it’s my favorite language 😊

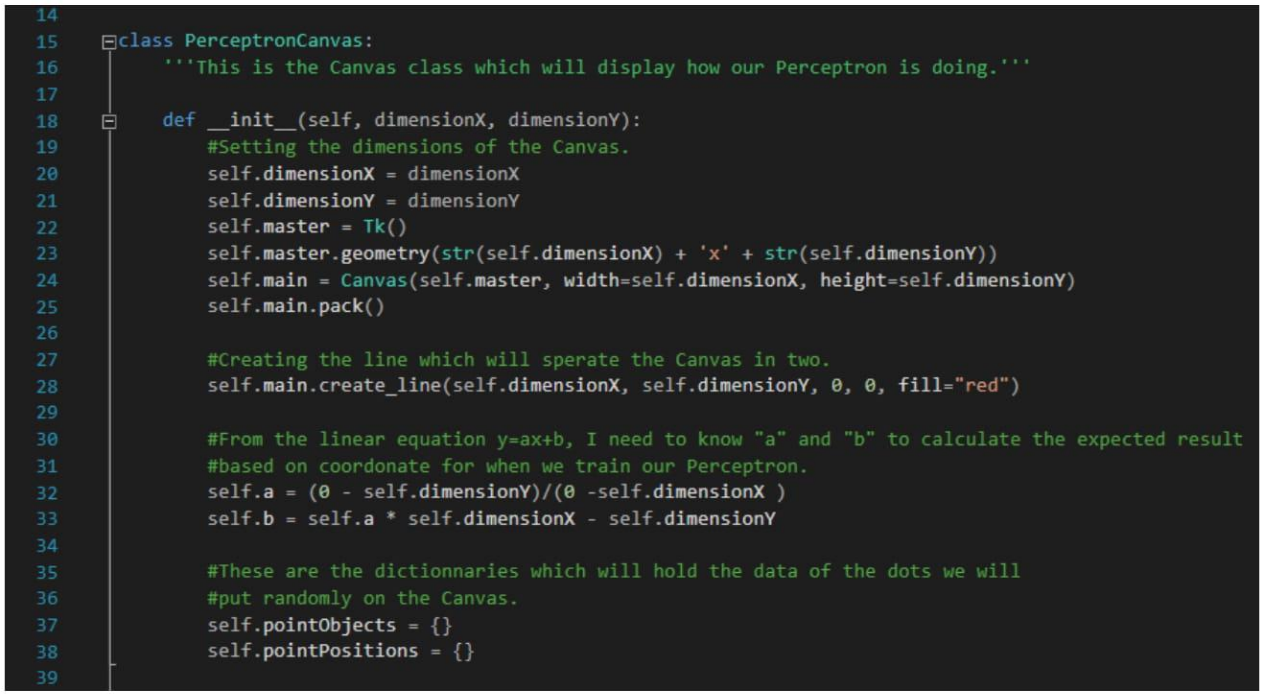
We first need to import the necessary libraries. First, I will import numpy, which is a library that can do matrices. Then, I will import Canvas (tkinter), which is a library that can create visual canvas (for visual demonstration). We will also need to function Randint from the Random module, which will allow us to generate a random number (we will need that to generate a random position for a given dot) and the Sleep function which will slow down our Perceptron, because otherwise he will learn so fast we wont be able to see anything.

To install the libraries,



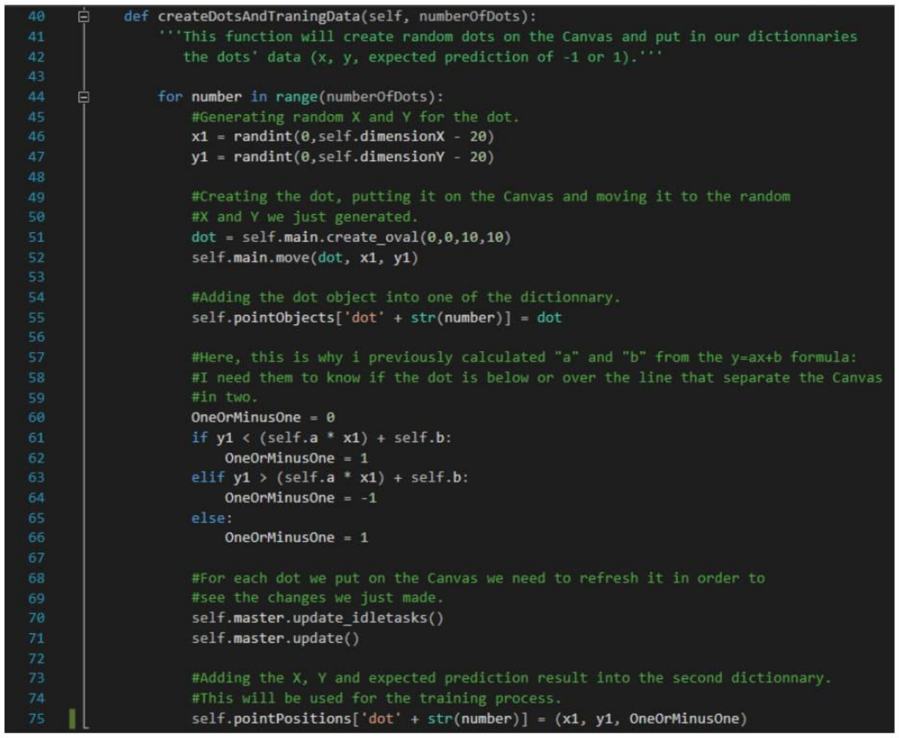
```python
#Numpy is a must-know in Python: If you do math, this library is your best friend.
import numpy

#This is the library we will use to draw the Canvas.
from tkinter import Canvas, Tk

#We need to generate random positions for the dots.
from random import randint

#Because our Perceptron will be damn fast, we need to slow it down a bit
#in order to be able to see how he is training.
from time import sleep

```

We will then define our PerceptronCanvas class which will display how our Perceptron is learning. Here is what happen when we will do a new instance of this class:



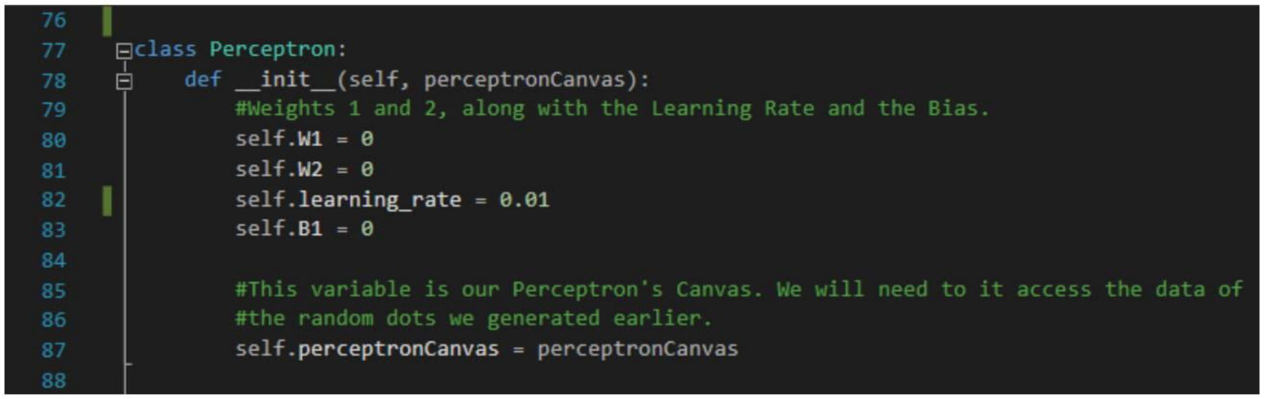
```python

class PerceptronCanvas:
    '''This is the Canvas class which will display how our Perceptron is doing.'''

    def __init__(self, dimensionX, dimensionY):
        #Setting the dimensions of the Canvas.
        self.dimensionX = dimensionX
        self.dimensionY = dimensionY
        self.master = Tk()
        self.master.geometry(str(self.dimensionX) + 'x' + str(self.dimensionY))
        self.main = Canvas(self.master, width=self.dimensionX, height=self.dimensionY)
        self.main.pack()

        #Creating the line which will sperate the Canvas in two.
        self.main.create_line(self.dimensionX, self.dimensionY, 0, 0, fill="red")

        #From the linear equation y=ax+b, I need to know "a" and "b" to calculate the expected result
        #based on coordonate for when we train our Perceptron.
        self.a = (0 - self.dimensionY)/(0 -self.dimensionX )
        self.b = self.a * self.dimensionX - self.dimensionY

        #These are the dictionnaries which will hold the data of the dots we will
        #put randomly on the Canvas.
        self.pointObjects = {}
        self.pointPositions = {}

```

The only method our PerceptronCanvas need is the one which will create dots, hold their data in memory so we can use it as training data for our perceptron, then draw the dots. That can be done in a single For loop. Here is the code:



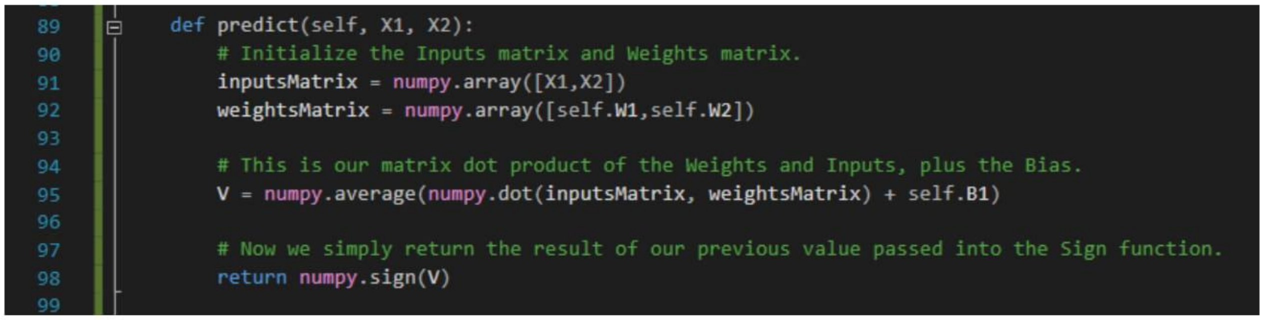
```python
    def createDotsAndTraningData(self, numberOfDots):
        '''This function will create random dots on the Canvas and put in our dictionnaries
           the dots' data (x, y, expected prediction of -1 or 1).'''

        for number in range(numberOfDots):
            #Generating random X and Y for the dot.
            x1 = randint(0,self.dimensionX - 20)
            y1 = randint(0,self.dimensionY - 20)

            #Creating the dot, putting it on the Canvas and moving it to the random
            #X and Y we just generated.
            dot = self.main.create_oval(0,0,10,10)
            self.main.move(dot, x1, y1)

            #Adding the dot object into one of the dictionnary.
            self.pointObjects['dot' + str(number)] = dot

            #Here, this is why i previously calculated "a" and "b" from the y=ax+b formula:
            #I need them to know if the dot is below or over the line that separate the Canvas
            #in two.
            OneOrMinusOne = 0
            if y1 < (self.a * x1) + self.b:
                OneOrMinusOne = 1
            elif y1 > (self.a * x1) + self.b:
                OneOrMinusOne = -1
            else:
                OneOrMinusOne = 1

            #For each dot we put on the Canvas we need to refresh it in order to
            #see the changes we just made.
            self.master.update_idletasks()
            self.master.update()

            #Adding the X, Y and expected prediction result into the second dictionnary.
            #This will be used for the training process.
            self.pointPositions['dot' + str(number)] = (x1, y1, OneOrMinusOne)
```

Now it's time to create our very first Perceptron class. Here, W1 is the Weight 1, W2 is the Weight 2 and B1 is the Bias. "perceptronCanvas" is the instance of our PerceptronCanvas class which hold the training data that will be generated when we will call the generating-dots method. Here is what will happen when we will create a new instance of our Perceptron class:



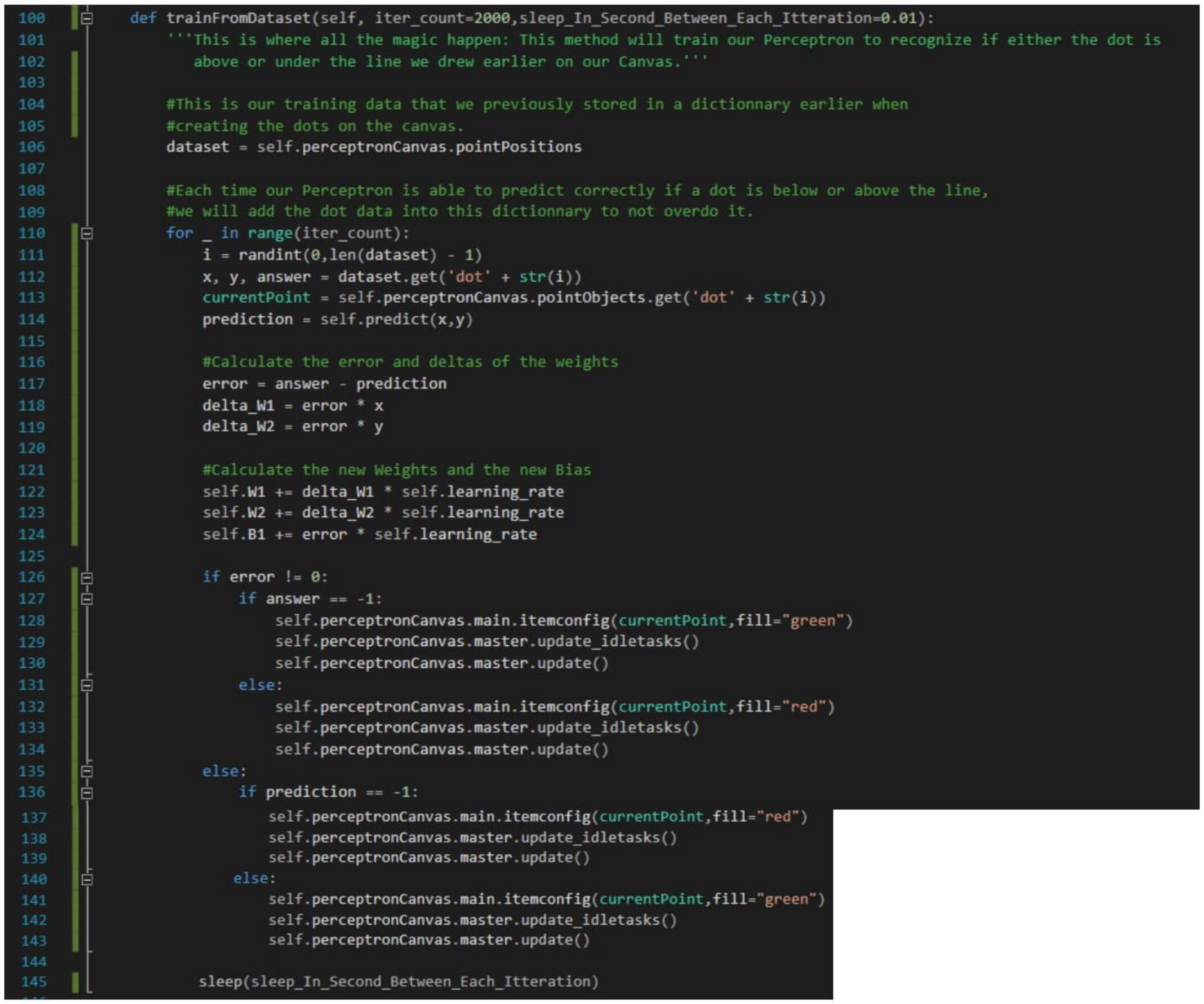
```python

class Perceptron:
    def __init__(self, perceptronCanvas):
        #Weights 1 and 2, along with the Learning Rate and the Bias.
        self.W1 = 0
        self.W2 = 0
        self.learning_rate = 0.01
        self.B1 = 0

        #This variable is our Perceptron's Canvas. We will need to it access the data of
        #the random dots we generated earlier.
        self.perceptronCanvas = perceptronCanvas

```

The next method will be the one that, given the inputs, will give us a prediction of either the dot is above or below the line. All we need to do is apply the formulas we talked about before and apply them. This is really simple using Numpy:



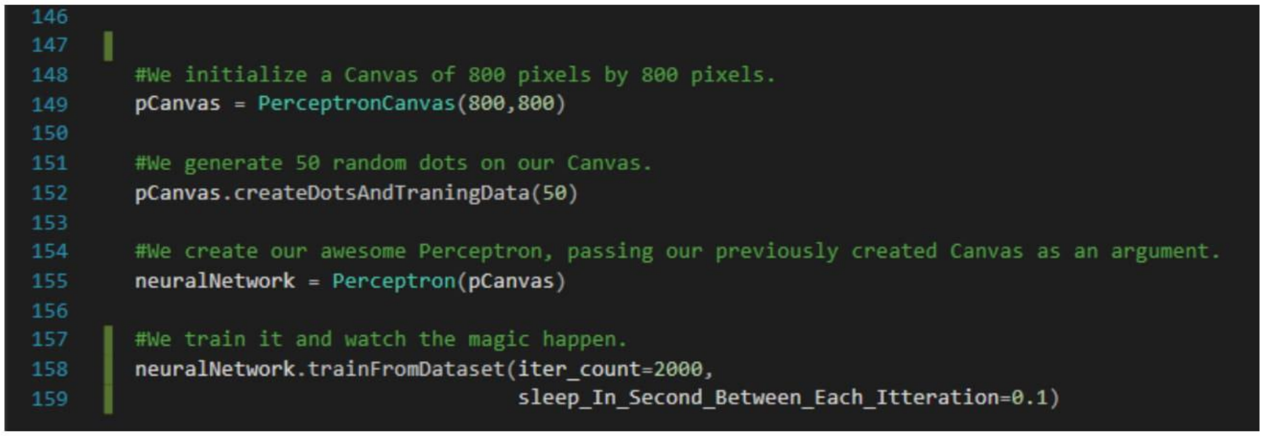
```python
    def predict(self, X1, X2):
        # Initialize the Inputs matrix and Weights matrix.
        inputsMatrix = numpy.array([X1,X2])
        weightsMatrix = numpy.array([self.W1,self.W2])

        # This is our matrix dot product of the Weights and Inputs, plus the Bias.
        V = numpy.average(numpy.dot(inputsMatrix, weightsMatrix) + self.B1)

        # Now we simply return the result of our previous value passed into the Sign function.
        return numpy.sign(V)

```

The final method of our Perceptron class will be the one that trains the Perceptron. We will refer to the training data that will be generated when we will create the dots. This method will have 2 parameters: the first one will be the number of iterations our Perceptron will guess if a given dot is below or above the line (we need that, because in a scenario where the Perceptron is complex and we cannot achieve a 100% success rate every time, we need to set a limit) and the second one is how long the Perceptron should wait before guessing the next dot, adjusting the weights and bias according to its guess, etc. Enough talking, here is the method:



```python
    def trainFromDataset(self, iter_count=2000,sleep_In_Second_Between_Each_Itteration=0.01):
        '''This is where all the magic happen: This method will train our Perceptron to recognize if either the dot is
           above or under the line we drew earlier on our Canvas.'''

        #This is our training data that we previously stored in a dictionnary earlier when
        #creating the dots on the canvas.
        dataset = self.perceptronCanvas.pointPositions

        #Each time our Perceptron is able to predict correctly if a dot is below or above the line,
        #we will add the dot data into this dictionnary to not overdo it.
        for _ in range(iter_count):
            i = randint(0,len(dataset) - 1)
            x, y, answer = dataset.get('dot' + str(i))
            currentPoint = self.perceptronCanvas.pointObjects.get('dot' + str(i))
            prediction = self.predict(x,y)

            #Calculate the error and deltas of the weights
            error = answer - prediction
            delta_W1 = error * x
            delta_W2 = error * y

            #Calculate the new Weights and the new Bias
            self.W1 += delta_W1 * self.learning_rate
            self.W2 += delta_W2 * self.learning_rate
            self.B1 += error * self.learning_rate

            if error != 0:
                if answer == -1:
                    self.perceptronCanvas.main.itemconfig(currentPoint,fill="green")
                    self.perceptronCanvas.master.update_idletasks()
                    self.perceptronCanvas.master.update()
                else:
                    self.perceptronCanvas.main.itemconfig(currentPoint,fill="red")
                    self.perceptronCanvas.master.update_idletasks()
                    self.perceptronCanvas.master.update()
            else:
                if prediction == -1:
                    self.perceptronCanvas.main.itemconfig(currentPoint,fill="red")
                    self.perceptronCanvas.master.update_idletasks()
                    self.perceptronCanvas.master.update()
                else:
                    self.perceptronCanvas.main.itemconfig(currentPoint,fill="green")
                    self.perceptronCanvas.master.update_idletasks()
                    self.perceptronCanvas.master.update()

            sleep(sleep_In_Second_Between_Each_Itteration)
```

Finally! Now all we have to do is call everything and watch 😊



```python


#We initialize a Canvas of 800 pixels by 800 pixels.
pCanvas = PerceptronCanvas(800,800)

#We generate 50 random dots on our Canvas.
pCanvas.createDotsAndTraningData(50)

#We create our awesome Perceptron, passing our previously created Canvas as an argument.
neuralNetwork = Perceptron(pCanvas)

#We train it and watch the magic happen.
neuralNetwork.trainFromDataset(iter_count=2000,
                               sleep_In_Second_Between_Each_Itteration=0.1)
```

Look at this beauty! You are now able to do a Perceptron neural network by yourself 😊