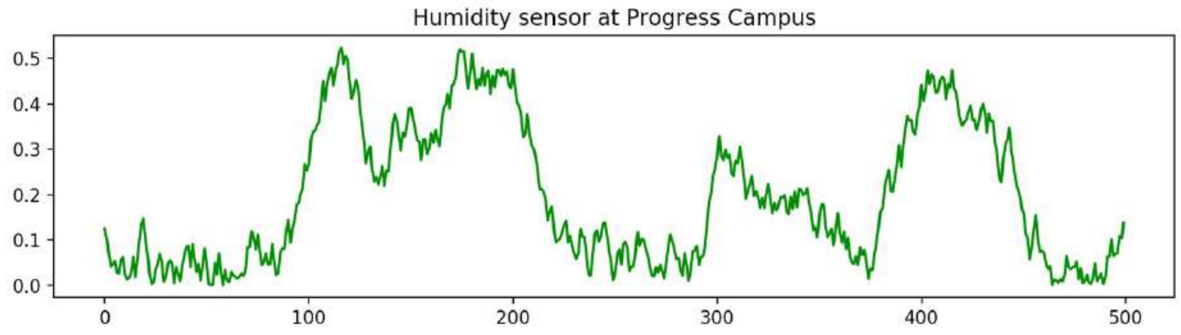
Networking for Software Developers

1. Lab 6 – Generator.

Most application depends on data. In IoT data is collected/generate by sensors. Since we do not have access to physical sensors, we will have to generate data via code. You will create a class with a member (property) that give you a “random” value. If you generate a large number of these values (at least 500) and plot it using matplotlib you should get the following diagram or something very similar:



Requirements:

2. You will pick a quantity that your value will mimic (such as earth tremors, temperature, cars passing through as intersection, humidity, barometric pressure, customers arriving at a mall, or just with an alternate descriptor). This will guide you when transforming the generated values to actual read-life data values to display on your diagram.
   You must have as idea on the range of value that will model your chosen quantity.

3. **Design and build a class** that will model your sensor reasonably well. Notice how the peaks do not occur at regular interval, nor are they the same height. Even the squiggles are not the same shapes.

4. Your class must have a fair amount of customization but at the same time should be easy to use, so provide a constructor with lots of default values.

5. Make it so that you can generate your data by repeated calling a method or accessing a property of the class instance. The member will give values ONLY in the range `[0, 1.0)`.
   Your generator must return a single float, not a list of floats.

6. Provide the code to display a diagram similar to the above which was done using matplotlib. You will have to convert the values obtained in step 4 to your range. You will provide meaningful label for the axis's and the title.

See the appendix of this document for some code sample and possible directions to explore. You will need some combination of the last three examples.

- Use `generator_4()` will give peaks and valleys
- Use `generator_3()` to change the length (or frequency) of the peaks.
- Use `generator_2()` and to get the squiggles.

DO NOT USE THE CODE AS IS!. Look at the intension behind the code.

Submission

1. Your code file will be named `group_7_lab6.py` .

Sample Code:

The following example illustrate the various possibilities in generating data.

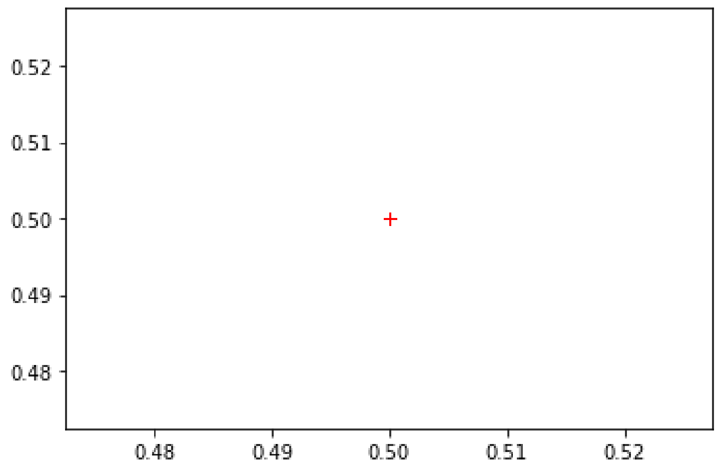
Constant value

The first example gives you a constant value regardless of how many times you call it.

```
import matplotlib.pyplot as plt

def generator_1() -> int:
    '''
    This is the greatest generator.
    It returns Narendra's favourite number
    '''
    return 0.5

number_of_values = 200
y = [generator_1() for _ in range(number_of_values)]
x = [generator_1() for _ in range(number_of_values)]
plt.plot(x, y, 'r+')
plt.show()
```



Uniform values

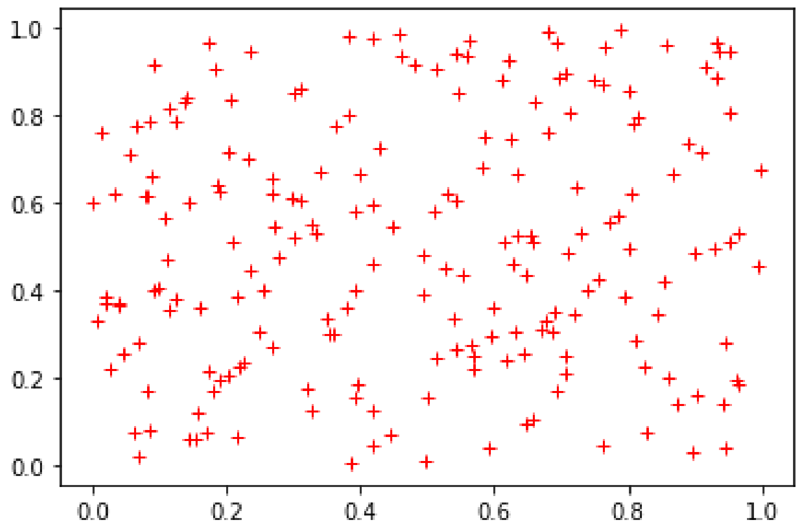
The second example gives you a uniformly random value. It uses the `randint()` method of the random class that returns a value in the interval `[a, b]`. Uniform distributed values occur frequently in everyday situations such as the odds of getting a particular value on the toss of an un-biased die.

```
import matplotlib.pyplot as plt
import random



def generator_2() -> int:
    '''
    This generator gives you a uniform random number in a 0 to 20
    '''
    return random.randint(0, 20)

number_of_values = 200
y = [generator_2() for _ in range(number_of_values)]
x = [generator_2() for _ in range(number_of_values)]
plt.plot(x, y, 'r+')
plt.show()
```



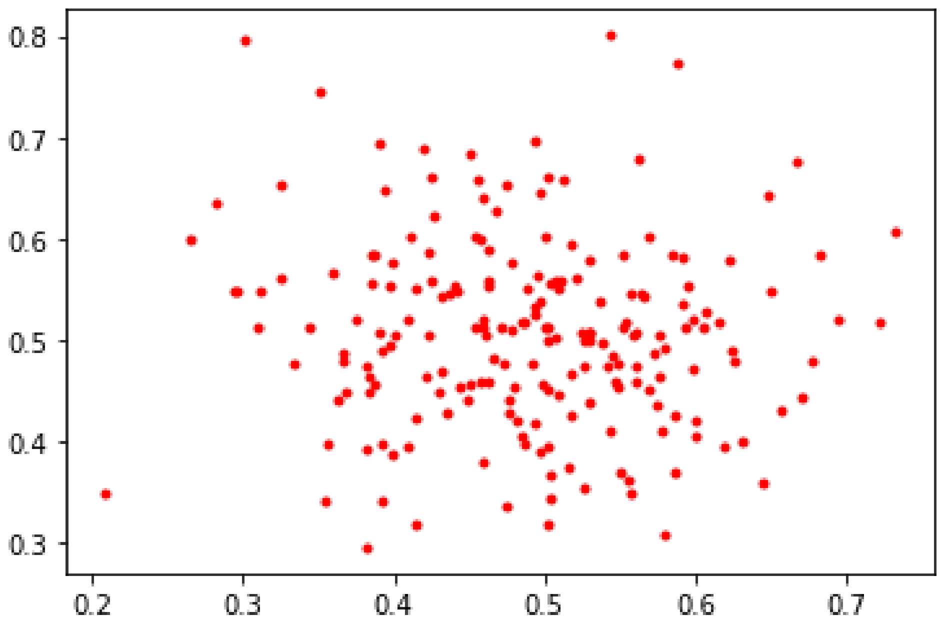
Standard values

The third example gives you a normal random value. It uses the `gauss()` method of the random class that returns a value based on a mean and a standard deviation. Normal/standard distributed values also occur frequently in everyday situations such as the number of students in a queue waiting for the TTC bus are the Progress terminal. The is different because the number of students in the line quickly builds up to a maximum when the bus has arrived and a minimum when there is no bus.

```python
import matplotlib.pyplot as plt
import random

def generator_3() -> int:
    return random.gauss(0.5, 1.0)

number_of_values = 200
y = [generator_3() for _ in range(number_of_values)]
x = [generator_3() for _ in range(number_of_values)]
plt.plot(x, y, 'r.')
plt.show()
```



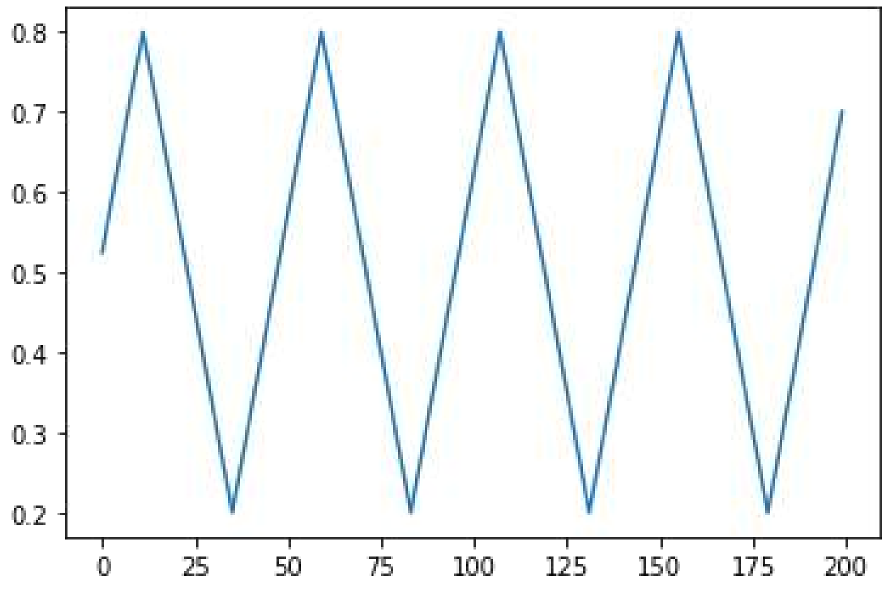
If you increase the number of points you will see there is a cluster at the center of the grid.

Pattern values

The fourth example gives you a saw-tooth wave form pattern. It starts at a value and goes up to a max, from there it comes down to a min and so on…

```
base = 0.5
min = 0.2
max = 0.8
delta = 0.25
def generator_4() -> float:
    global: base, min, max, delta
    if base < min base > max:
        delta *= -1
    base += delta
    return base

number_of_values = 200
y = [generator_4() for _ in range(number_of_values)]
plt.plot(y, 'g')
plt.show()
```

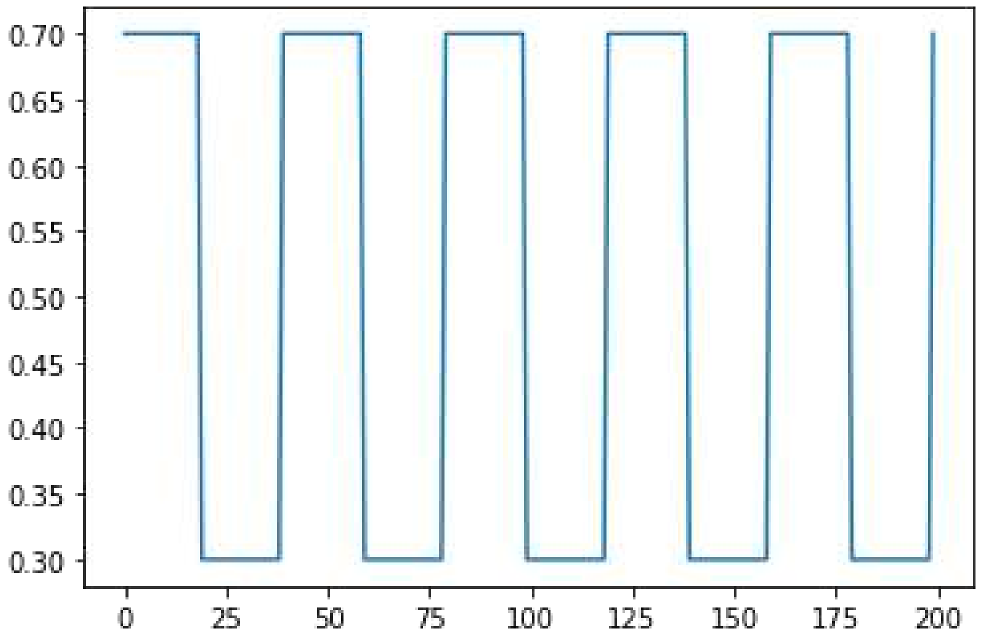


Pattern values

This fifth example gives you a value that follows a square-shaped pattern.

```
period = 20
delta = 0.2
def generator_6() -> float:
    global period, delta
    period -= 1
    if: period == 0:
        delta *= -1
        period = 20
    return 0.5 + delta


number_of_values = 200
y = [generator_6() for _ in range(number_of_values)]
plt.plot(y)
plt.show()
```


0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0
25
50
75
100
125
150
175
200