Implementation Of Internet of Things with Python

Generally, prototypes or real-life Internet of Things (IoT) systems have to be designed and developed swiftly and competently. Whenever this occurs, two activities instantly come to life: One is to program the IoT devices, and another is to organize a backend to interact with these devices.

In both activities, we can utilize the Python programming language for their development. Or we can utilize a functional and practical edition of MicroPython in order to work on devices with small computing resources, and accordingly, at a very low price.

In the following tutorial, we will understand the use of Python in programming Internet of Things (IoT) devices and create a backend for them to work.

But before we get to that, let us briefly discuss the importance of IoT.

Understand the importance of the Internet of Things

The term "**Internet of Things**" was first coined in the year 1999 by **Kevin Ashton**. Ever since the importance and scale of IoT have exploded, one of the chief indicators is that the market size of the IoT was $151 billion in 2018, with a steady increment year after year. As per the predictions of marketers, the IoT market could cross the $561 billion mark by 2022.

Back in the day, we could explain IoT with examples as shown below:

"We can utilize the phone to turn a light bulb on and off in the room."

Nowadays, hardly anyone would be amazed by a smart electricity meter that transmits readings of the consumption of the electricity, uploads that information to the cloud, and produces monthly bills sent directly to the e-mail.

IoT is increasingly utilized across industries in order to streamline processes and make them more efficient. For instance, manufacturing production lines and agriculture are great examples of various industries taking benefit of the different features of IoT. In the particular scenario of agriculture, IoT helps in coordinating harvesters with trucks that have elevators to handle grains efficiently.

Why use Python in the Internet of Things?

For many developers, Python is considered as the language of preference in the market. It is simple to learn, has clean syntax, and has a large online community supporting it. Python becomes a great choice when it comes to IoT. We can either use it for the backend side of development or the software development of devices. Moreover, Python is available to work on Linux devices, and we can make use of MicroPython for microcontrollers.

Python is the coding language that we can use to reduce the volume of data that we need to deal with, accessible in the cloud. Python recognizes the needs regardless of whether we create the IoT project from scratch or interact with actuators, sensors, and accessories.

Some of the many benefits of working with Python for IoT devices are a large number of libraries for all types of platforms and the speed it offers at which we can develop the code.

Python is a great ally for developing device prototypes. Even if we rewrite some of the scripts while producing to C, C++ or Java to improve performance, the system will generally function perfectly in Python.

What are the best solutions for IoT in Python?

Some of the best solutions for IoT in the Python programming language are as follows:

1. Python on Raspberry Pi
2. Python on PyBoard
3. ESP8266, ESP32 with Micropython

We will discuss each solution in brief.

Python on Raspberry Pi

The primary objective of running Python on an IoT device that pops up in mind is grabbing the Raspberry Pi from the table. Python is pre-installed in the operating system, and the only objective left for us is to write the coding script.

In this scenario, we can control the I/O ports on the expansion bar of the Raspberry Pi. Fortunately, the board supports wireless communication (Bluetooth and WiFi) and Ethernet. We can also connect a monitor to the HDMI output, a specialized 3.2" 320x240 TFT LCD, or a low energy consumption E-Ink 2.13" 250x122 display for Raspberry Pi.

There are controllers available in a large variety of computing power and budgets. We can choose these controllers for the IoT system - ranging from the fast Raspberry Pi 4 Model B 8 GB to the smallest Raspberry Pi Zero, all supporting the Python programming language. In case of necessity, we can install the earlier version of Python 2.7 for past compatibility.

Let us consider the following snippet of Python code where we have used the GPIO Zero library in order to control the I/O ports.

Example:

```
# importing the required modules
from gpiozero import Button
from time import sleep

# creating an object of Button
the_button = Button(2)

# using the if-else statement
while True:
    if the_button.is_pressed:
        print("Button Pressed")
    else:
        print("Button Released")
    sleep(1)
```

Explanation:

The above example demonstrates the receiving and processing of the signals by pressing the button on the second pin at the moment of release.

The benefits of utilizing this approach are the availability of a large variety of development utilities, libraries and communications for the most complex devices based on Raspberry Pi involving video processing from cameras.

Python on PyBoard

Another great solution for Python in IoT devices is the PyBoard with an STM32F405RG microcontroller.

The PyBoard is considered a compact as well as a powerful electronics development board. It works on MicroPython. The PyBoard connects to the PC through USB, providing us with a USB flash drive to store the Python scripts and a serial Python prompt (a REPL) for instant programming. This works with Windows, MacOS, and Linux.

PyBoard executes MicroPython, which is a lightweight implementation of the standard CPython interpreter. The official documentation also says: "MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimized to run on microcontrollers and in constrained environments. The MicroPythonpyboard is a compact electronic circuit board that runs MicroPython on the bare metal, giving you a low-level Python operating system that can be used to control all kinds of electronic projects. MicroPython is packed full of advanced features such as an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling and more. Yet it is compact enough to fit and run within just 256k of code space and 16k of RAM."

MicroPython is an entire rewrite of the Python (version 3.4) programming language to fit and execute on a microcontroller. It involves various optimization for efficiency and consumes quite less RAM.

MicroPython executes bare-metal on the PyBoard, necessarily providing us with an Operating System based on Python. The in-built pyb module consists of functions and classes in order to control the peripherals available on the board, like I2C, UART, ADC, DAC, and SPI.

The board's dimensions are impressive, taking up around two quarters, 33mm x 43mm and weighing only 6 grams.

ESP8266, ESP32 with MicoPython

Another option could be using ESP8266 and ESP 32 to run Python. We have to create a device based on the Internet of Things with low power consumption, great capabilities, and integration with wireless Wi-Fi networks. More precisely, we can use MicroPython.

Once we installed Python on the system, we can use the pip installer in the command line in order to install the **esptool** module. The syntax for the same is shown below:

Syntax:

1. $ pip install esptool

The installation procedure of the MicroPython is pretty easy. We can download the firmware from the website and install it with the help of esptool, not forgetting to format the board before installing it.

We can also use one of the IDEs used for developing with MircoPython. The complete procedure of development is carried out on a working computer, and then it is compiled and saved in the memory of an ESP8266 or ESP32 microcontroller.

Let us consider the following example to see how simple the script might look like:

Example:

```
# importing the required modules
from machine import Pin
import time

# creating an object of Pin
ledPin = Pin(2, Pin.OUT)

# using some functions
while True:
    ledPin.on()
    time.sleep(1)
    ledPin.off()
    time.sleep(1)
```

Explanation:

In the above snippet of code, we have imported the **Pin** module from the **machine** library along with the **time** module. We have then created an object of **Pin** and execute some functions on it.

MicroPython imposes many restrictions compared to regular Python; however, in general, we can easily write the necessary functionality on the client-side and execute it effectively on ESP microcontrollers. This option is relatively more cost-effective than buying PyBoard.

Understanding the use of Python in IoT Backend

We can use Python as a Backend programming language for the Internet of Things in many ways. Some of them are as follows:

MQTT protocol with Python

One of the most popular connection methods for IoT devices is MQTT, and it is a protocol used for effective implementation with Python.

The MQTT protocol is a machine-to-machine (M2M)/Internet of Things connectivity protocol designed as a highly lightweight publish/subscribe messaging transport. It is used to connect to remote locations where a small code footprint is needed, and network bandwidth is premium.

The Python client library called **Eclipse Paho MQTT** implements versions 3.1, 3.1.1, and 5.0 of the MQTT protocol.

The code of the **Poho** library offers a client class that allows applications to link to an MQTT broker in order to publish messages, subscribe to topics, and receive published messages. It also delivers some helper functions to make things simpler in publishing one-off messages to MQTT servers.

Moreover, this library supports Python 2.7.9 and above or 3.5 and above. The integration of images with older 2.7 versions of Python is straightforward.

IoT backend on Flask in Python

We can also use the Flask microframework to write the backend for the IoT systems. The Flask microframework is a quick and hassle-free tool that easily set up server-side I/O information, and it is also packed with many functionalities that make work more efficient.

We can start by deciding the requests we have to serve from the IoT devices. We then have to set up the Flask microframework and write a block of code. The **GET** method will then return information as per the request from the side of the client.

In several cases, we are best off focusing on the RESTful protocol while working with the IoT devices. This allows us to simplify the exchange between the components of the system and helps us to expand the system of exchanging information in the future.

Let us consider a task that has arisen as follows: Display information from IoT devices on a web page. The Flask microframework will rescue us again with its core template mechanism where we can design the required web page with the data display involving graphics.

The disadvantage of utilizing this method is the potential lack of starting the data transfer from the server to the device. Thus, the IoT must periodically and independently pull from the server. Rest easy, as there are keys to report this risk. We can utilize web sockets or a Python library for Pushsafer. PushSafer is an easy and safe way to send and receive push notifications in real-time to Android, iOS, and Windows devices (mobile as well as desktop), including internet browsers such as Google Chrome, Mozilla Firefox, Opera, and many more.

Microsoft Azure IoT backend in Python

Microsoft has released a new open-source extension for IoT to extend the capabilities of Azure CLI 2.0. Azure CLI 2.0 involves commands to interact with the Azure Resource Manager and endpoints of management.

For instance, we can utilize Azure CLI 2.0 to build an Azure Virtual Machine or IoT Hub. The extension of CLI allows an Azure service to complement Azure CLI by providing users access to additional capabilities specified to services.

The Extension of IoT offers programmers command-line access to the capabilities of the IoT Edge, IoT Hub and IoT Hub Device Provisioning Service.

Azure CLI 2.0 allows instant management of resources of Azure IoT Hub, devices provisioning services instances, and associated hubs. The new IoT extension enriches Azure CLI 2.0 with features such as device management and all IoT Edge capabilities:

1. Azure CLI 2.0 IoT capabilities - Control Plane
2. Managing instances of IoT Hub, consumer - groups and jobs
3. Managing instances of device provisioning service, access-policies, Linked - hub and certificates
4. New features for extensions - data plane
5. Managing device and edge module identities and their respective twin definitions
6. Querying IoT Hub for details like device and module jobs, twins, and messaging routing
7. Invoking methods of device and module
8. Generating SAS tokens and grabbing connection strings
9. Cloud-to-device and Device-to-cloud messaging
10. Device file uploading
11. Device simulation for testing

Amazing hacks of Python

In this tutorial, we will learn how awesome a Python language is for coding. We will discuss some of its amazing hacks, making Python the best among other languages.

Hacks of Python

Following are some amazingly cool hacks of Python that can make work easy for users and developers:

1. List Comprehensions: It is the best and efficient technique for getting rid of writing pointless lines of program.

List comprehension consists of the following parts:

Output expression
Input sequence
A member of the input sequence represented by a variable
The optional predicate parts.
Example:

```
import functools as FT

# First, filter odd numbers
list_1 = filter(lambda K : K % 2 == 1, range(10, 30))
print ("List: ", list(list_1))

# Then we will filter the odd square which is divisible by 5
list_1 = filter(lambda K : K % 5 == 0,
     [K ** 2 for K in range(1, 11) if K % 2 == 1])
print ("ODD SQUARE WHICH IS DIVISIBLE BY 5: ", list(list_1))

# Here, we will filter negative numbers
list_1 = filter((lambda K : K < 0), range(-10, 10))
print ("Filter negative numbers: ", list(list_1))
```

```
# Now, implement by using the max() function
print ("Maximum Number in the List: ")
print (FT.reduce(lambda S, T: S if (S > T) else T, [14, 11, 65, 110, 105]))
```

Output:

```
List:  [11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
ODD SQUARE WHICH IS DIVISIBLE BY 5:  [25]
Filter negative numbers:  [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
Maximum Number in the List:
110
```

2. Printing a List: Lists are not printed according to our requirements; they are always printed in unnecessary square brackets and single quotes. But in Python, we have a solution for printing lists efficiently by using the join method of string. The "join method" can turn the list into a string by classifying every item into a string and connecting them with the string on which the join method is used.

Example:

```
# First declare the list:
ABC = ['LPG', 'WWF', 'XYZ', 'MPG']

# Then, we will print the list:
print ("The Simple List: ", ABC)

# HEre, we will Print the list by using join method
print ('The List by using join method: %s' % ', ' .join(ABC))

# we can directly apply Join Function on the List:
print ('Directly applying the join method: ', ('', '' .join(ABC)))
```

Output:

```
The Simple List:  ['LPG', 'WWF', 'XYZ', 'MPG']
The List by using join method: LPG, WWF, XYZ, MPG
Directly applying the join method:  LPG, WWF, XYZ, MPG
```

3. Transpose a Matrix: In Python, a user can implement the matrix as a nested list, which means a list inside a list. Every element of the list is treated as a row of the matrix.

Example:

```
M_1 = [[5, 3], [1, 2], [9, 8]]
print ("Matrix 1: ")
for row in M_1 :

    print (row)
rez_1 = [[M_1[K][L] for K in range(len(M_1))] for L in range(len(M_1[0]))]
print ("\n")
print ("Matrix 2: ")
for row in rez_1:

    print (row)
```

Output:

Matrix 1:
[5, 3]
[1, 2]
[9, 8]

Matrix 2:
[5, 1, 9]
[3, 2, 8]

4. artition of List into "N" Groups: The users can use the iter() function as an iterator over the sequence.

Example:

```
# First, we will Declare the list:
```

```
LIST_1 = ['E_1', 'E_2', 'E_3', 'E_4', 'E_5', 'E_6']

partition_1 = list(zip (*[iter(LIST_1)] * 2))
print (''List after partitioning into different of groups of two elements: '', partition_1)
```

Output:

List after partitioning into different of groups of two elements: [('E_1', 'E_2'), ('E_3', 'E_4'), ('E_5', 'E_6')]

Explanation:

In the above code, we used ''[iter(LIST_1)] * 2'' which produced different groups containing two elements of the 'LIST_1[]' list. That is, the lists of length two will be generated using the elements from the first list.

AD

5. Print more than One Item of List simultaneously

Example:

```
list_1 = [11, 13, 15, 17]
list_2 = [12, 14, 16, 18]

# Here, we will use zip() function which will take 2 equal length list
# and then merge them together into pairs
for K, L in zip(list_1, list_2):
    print (K, L)
```

Output:

11 12
13 14
15 16
17 18

6. Take the String as Input and Convert it into List:

Example:
AD

```
# Reading a string from input as int format
# after splitting it's elements by white-spaces
print ("Input: ")
formatted_list_1 = list(map (int, input().split()))
print ("Output as Formatted list: ", formatted_list_1)
```

Output:

Input:
10 12 14 16 18 20 22
Output as Formatted list: [10, 12, 14, 16, 18, 20, 22]

7. Convert List of Lists into Single List:

Example:

```
# importing the itertools
import itertools as IT

# Declaring the list geek
LIST_1 = [[1, 2], [3, 4], [5, 6]]

# chain.from_iterable() function will return the
# elements of nested list
# and iterate it from first list
# of iterable till the last
# end of the list

list_2 = list(IT.chain.from_iterable(LIST_1))
print ("Iterated list of 'LIST_1': ", list_2)
```

Output:

Iterated list of 'LIST_1': [1, 2, 3, 4, 5, 6]

8. Print the Repeated Characters: Suppose our task is to print the patterns like "122333444455555666666". We can easily print this pattern in Python without using for loop.

Example:

```
print ("1" + "2" * 2 + "3" * 3 + "4" * 4 + "5" * 5 + "6" * 6)
```

Output:

122333444455555666666