Micropython is an implementation of Python 3 that aims microcontrollers and embedded devices. Given the processing and memory limits of such devices, micropython only includes a small subset of the Python standard library. It also ships with libraries to control typical components of microcontrollers such as ADC, GPIO, SPI, I2C and UART.

OFFICIAL DOCUMENTATION

The official documentation of micropython is located here -> http://docs.micropython.org/en/latest/pyboard/

Python is an interpreted programming language, featuring structured and object-orientated programming, as well as some functional programming capabilities. It has the particularity to be both strongly and dynamically typed. This basically means that it has strict rules regarding memory management, but can handles them automatically (they are mostly transparent to the programmer).

Note that the standard extension for Python files in *.py. We provide bellow some code examples for basic structures in Python.

OFFICIAL DOCUMENTATION

The official documentation of Python 3 is located here -> https://docs.python.org/3/reference/

TRY IT YOURSELF!

If you properly installed the software described here (or use our virtual machine), you can use the Spyder3 IDE to test pieces of codes and sharpen your skills in Python!

ADDING COMMENTS

```
    #Comments are preceded with a sharp.
```

IMPORTING A MODULE

Modules in Python are the equivalent of libraries in most other programming language. A module stores a collection of class and/or functions, that we can conveniently call from our program using the keyword import.

```
import numpy
```

Sometimes, modules are grouped together in packages.

```
from os import *    #Import ever module from package os

from os import I2C #Import the module I2C of package os
```

Once a module has been imported, we can call its elements using the following syntax:

```
import numpy


numpy.zeros(2) #We call function zeros() of module numpy
```

VARIABLE DECLARATION

Variables can store a variety of data: strings, integer, floating point values, lists, etc. A value is assigned to a variable using the keyword "=", and this this value can be changed anytime.

PYCOM'S MICROPYTHON DOCUMENTATION

The micropython implementation shipped in the LoPy 4 slightly differs from the official one. Documentation for Pycom's implementation of micropython is located here -> https://docs.pycom.io/firmwareapi/introduction

TRY IT YOURSELF!

You can use this online simulator (http://micropython.org/unicorn/) to test pieces of codes and sharpen your skills in micropython!

THE REPL

The "Read-Eval-Print-Loop" is an interactive terminal that lets you interact with your LoPy in Python in real-time. It is very convenient to test pieces of code that may ultimately got to a Python file.

The REPL is characterized by its ">>>" prompt.

```
>>>
```

BOOT.PY AND MAIN.PY

boot.py and main.py are two Python scripts that are executed on boot. You may open and study them, but our advice is not to modify them, as they have been written to ease your developments.

boot.py is configured to stop the SDS011 fan on boot (in case it would be spinning). It will also turn-off the WiFi access-point when not in programming mode, as a way to save battery.

main.py handles the automatic call of iiot_full_sensor.py script when the sensor is not in programming mode. It also handles reboot upon catching an unexpected exception.

MAIN LOOP

As micropython aims microcontroller programming, main programs often feature a so-called main loop, or infinite loop. This loop simply contains the sequence of instruction that your LoPy is supposed to execute. It is however advised to declare variables and objects before this loop, to avoid wasting time reallocating and initializing resources at each loop iteration.

```
#Global variable and objects declaration here.


while True:

    #Your instructions here.
```

RGB LED

The LoPy features an on-board RGB LED. To use it you must import the rgbled library from package pycom:

```
from pycom import rgbled
```

Then, you choose the color to display using the following command:

```
rgbled(0xRRGGBB)
```

where:

- RR is the amount of red between 00 and FF
- GG is the amount of green between 00 and FF
- BB is the amount of blue between 00 and FF.

```
rgbled(0xFF0000) #Full red

rgbled(0x00FF00) #Full green

rgbled(0x0000FF) #Full blue

rgbled(0xFFFFFF) #Full white

rgbled(0x000000) #LED off
```

I2C BUSES

I2C is a protocol used when a "master" device wants to communicate with many "slave" devices. It needs two wire: one the for data transfer, one for the clock reference.

In Python, it is configured in the following way:

```
from machine import I2C

â€‹

i2c = I2C(0, pins=('P10','P11'))      #Use non-default PIN assignments (here with P10=SDA and P11=SCL) on bus 0

i2c.init(I2C.MASTER, baudrate=100000) #LoPy is the master, baudrate must be less than 400000 bits/s


i2c.scan() #Scan for slave devices on the bus, and report their adresses.
```

You can use up to 3 I2C buses on the LoPy. The identifier of the I2C bus that you want to use (0, 1 or 2) is given as first argument of the I2C constructor.

UART (SERIAL) CONNECTIONS

A Universal Asynchronous Receiver-Transmitter is used for communication between exactly two devices. Physically, it uses two wires: one carries data from device 1 to device 2, the other carries data from device 2 to device 1.

In Python, it is configured in the following way:

```
from machine import UART

â€‹

#Init UART 0 with a baudrate of 9600 bauds and use non-default PIN assignments (here with P20=TX and P21=RX).

uart = UART(0, 9600, pins=('P20','P21'))
```

You can use up to 3 UART links on the LoPy. The identifier of the UART bus that you want to use (0, 1 or 2) is given as first argument of the UART constructor.

DIGITAL INPUT/OUTPUTS

Digital inputs and outputs are controlled with objects of type Pin from package machine:

```
from machine import Pin

â€‹

p10 = Pin('P10', mode=Pin.OUT) #Configure pin P10 as output

p11 = Pin('P12', mode=Pin.OUT) #Configure pin P10 as input


p10.value()     #Gives the logical level measured on pin 10.

p11.value(True) #Sets the logical level of pin 11 to 1.

p11.value(False) #Sets the logical level of pin 11 to 0.
```

ANALOG INPUTS (ADC)

The LoPy is equipped with a 12-bit ADC capable of measuring analog voltages from 0.0V to 1.1V (attenuators can be set in front of the ADC to extend the range of measurable voltages). Digital inputs and outputs are controlled with objects of type ADC from package machine:

```
from machine import ADC

â€‹adc = machine.ADC()                                  #Create an ADC object

apin = adc.channel(pin='P16', attn=ADC.ATTN_0DB)     #Configure P16 as an analog input with 0dB of attenuation

apin = adc.channel(pin='P16', attn=ADC.ATTN_2_5DB)   #Configure P16 as an analog input with 2.5dB of attenuation

apin = adc.channel(pin='P16', attn=ADC.ATTN_2_5DB)   #Configure P16 as an analog input with 6dB of attenuation

apin = adc.channel(pin='P16', attn=ADC.ATTN_2_5DB)   #Configure P16 as an analog input with 11dB of attenuation


apin()         #Returns analog value measured at pin 16 (between 0 and 4096)

apin.voltage() #Returns voltage measured at pin 16 (takes into account the attenuation)
```

LORA

The LoPy is shipped with a LoRa transceiver, and the software stack is able to use the LoRaWAN MAC layer. LoRa physical parameters are handled by the library LoRa (in the network package), while LoRaWAN is handled by socket objects.

This is how a LoRaWAN link is configured:

```
from network import LoRa

import socket


###Initialize LoRa PHY###

lora = LoRa(mode=LoRa.LORAWAN)
```

```
#Power mode may be LoRa.ALWAYS_ON, LoRa.TX_ONLY or LoRa.SLEEP.

lora.power_mode(LoRa.TX_ONLY)


#Credentials

app_eui = b'\x48\x62\x66\x27\x56\x63\x68\x53'

app_key = binascii.unhexlify('11 22 33 44 55 66 77 88 11 22 33 44 55 66 77 88'.replace(' ',''))


#Join the network (or re-join if connection lost)

lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)


###Initialize LoRa PHY###

s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)


#Configuring data rate (between 0 and 5)

s.setsockopt(socket.SOL_LORA, socket.SO_DR, 3)


#Selecting confirmed/non-confirmed type of messages

s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, True) #Activate acknoledgements

s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, False) #Deactivate acknoledgements
```

A high data rate means more data per packet, but less resiliency to channel conditions. A lower data rate implies the opposite.

This is how a message is sent with this configuration:

```
#Wait for Lora connection

while not lora.has_joined():

    print('.', end='', flush=True)


#Prepare message

msg = 'Hello, world!'


#Wait until data is sent (max. 10s)

s.setblocking(True)

s.settimeout(10)


#Send message

try:

    s.send(msg)

except:

    print('Failed to send message!')
```