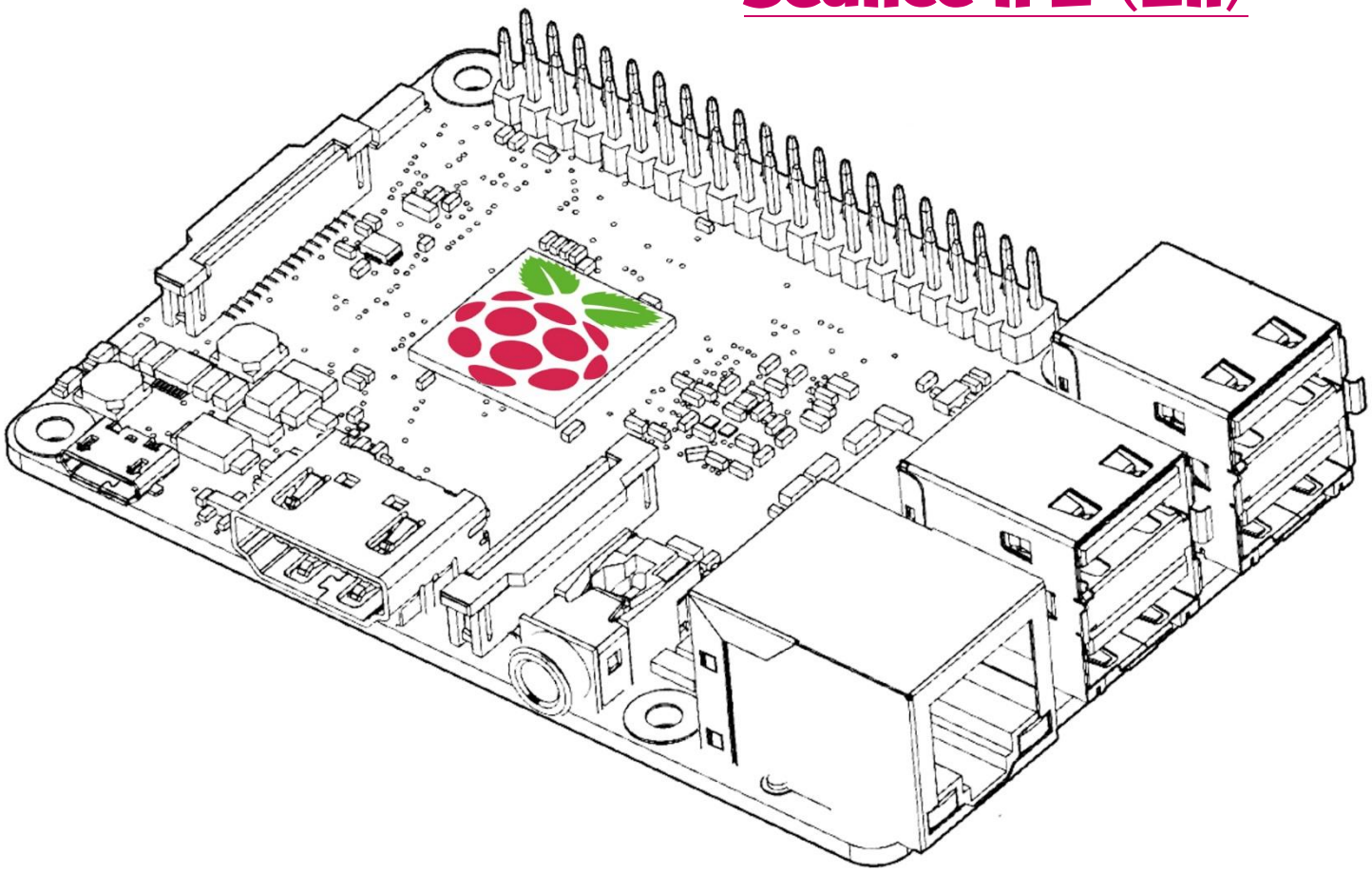
AT31 - Module IPE

IUT RENNES

Séance n°2 (2h)



Découverte des classes en Python

Michaël BOTTIN

Objectifs de la séance :

- Découvrir la carte d'extension 'vumètre'
- Comprendre la notion d'objet et de classe d'objet
- Ecrire sa première classe en Python
- Utiliser une librairie extérieure pour la gestion du capteur VL6180x

Récupération des fichiers utiles pour cette séance :

Comme pour le TP précédent, il vous faut récupérer les fichiers utiles pour l'écriture de vos programmes. Vous allez les récupérer depuis **Github**.

Pour cela, tapez la commande suivante dans le terminal :
$ git clone https://github.com/MbottinIUT/IUT_IPE_TP2.git
[remarque : il n'y a pas d'espace dans ce lien, juste des '_']

Une fois la tâche effectuée, vous devriez retrouver, en ouvrant un explorateur de fichiers, un dossier '**IUT_IPE_TP2**' sous '**/home/pi**'.

Notion d'objets et de classes :

A travers les exercices en Python du premier TP, vous avez pu aborder :

- L'affichage de texte et de variables
- La saisie d'une donnée utilisateur
- L'utilisation d'une boucle
- Le principe de la gestion des erreurs
- La création de fonctions
- L'utilisation d'une librairie externe (en fait une classe !!)

Toutefois, si ces exercices vous ont permis d'écrire vos premiers programmes, ils ne vous ont pas permis d'imaginer pourquoi Python est un langage si populaire à l'heure actuelle. En effet, ce que vous avez écrit en Python aurait pu être écrit en C mise à part la synthèse vocale.
On va donc essayer, avant d'aller plus loin, de présenter très rapidement ce qui caractérise un langage orienté objet comme le Python : les objets et les classes.
En effet, l'utilisation de classes et d'objets rendent un langage comme Python beaucoup plus puissant que le langage C par exemple.
Encore une fois, ce module n'est qu'une première approche directement par la pratique et ne sera pas suffisant pour bien comprendre la programmation orientée objet. Il faudrait un module beaucoup plus conséquent et un cours associé.

Qu'est-ce qu'un objet ?

Et bien, tout est 'objet' : un signal électrique, une voiture, un compte bancaire, un nombre complexe, un plan de bâtiment, …

En POO (Programmation Orientée Objet), l'idée est de représenter de manière simplifiée des objets du monde réel (concrets ou abstraits) avec des lignes de code.

Tout objet possède donc des caractéristiques, ce que l'on nomme '**Attributs**' en POO :

- Peugeot 508 : motorisation, boîte de vitesse, couleur, habillage intérieur, …
- Nombre complexe : partie réelle, partie imaginaire
- Compte bancaire : numéro, détenteur, solde, …

Dans un programme, toutes les caractéristiques réelles d'un objet ne sont pas forcément utiles. On n'attribuera à l'objet informatique que les caractéristiques utiles.

Mais un objet ne se décrit pas uniquement par ses caractéristiques, il se décrit également par des actions ou des réactions, ce que l'on nomme '**Méthodes**' en POO :

- Peugeot 508 : démarrer, s'arrêter, rouler, ouvrir hayon, régler rétroviseur, …
- Nombre complexe : déterminer son module, déterminer sa phase, …
- Compte bancaire : Afficher solde, effectuer un retrait, effectuer un dépôt, virement vers un autre compte, …

Et qu'est-ce qu'une classe ?

C'est un abus de langage de dire qu'une classe et un objet sont identiques. La classe est un peu le 'moule' permettant de créer des objets similaires. On dit qu'un objet est une **instance** d'une classe.
L'objet instancié disposera de tous les attributs et les méthodes de la classe, mais chaque objet instancié peut avoir des valeurs d'attributs différents.
Prenons l'exemple de la classe 'compte bancaire', on peut créer autant d'objets 'comptes' que l'on veut :

- Compte numéro 1078654491 – détenteur : Armand Dupont – nature : courant – solde : 1025,87€
- Compte numéro 1079635122 – détenteur : Cécile Nivers – nature : Livret A – solde : 41.023,75€

Quel que soit le compte, il hérite des différentes méthodes fournies par la classe 'compte bancaire' (effectuer un retrait, afficher le solde, …) mais chaque objet créé depuis cette classe peut avoir des attributs différents.

Généralement, on représente graphiquement une classe de la manière suivante :

Nom de la classe
Attributs
Méthodes()

Compte bancaire
numéro détenteur solde
afficher_solde() retrait() dépôt()

Nombre complexe
partie_réelle partie_imaginaire
module() phase()

Remarque : toute classe d'objet dispose forcément soit d'attributs, soit de méthodes, soit des deux. Mais l'un des deux peut être absent.

Remarque : les méthodes sont associées à des parenthèses '**()**', car elles s'apparentent à des fonctions. Elles peuvent donc avoir des paramètres si nécessaire.

Concrètement, comment accède-t-on aux attributs et aux méthodes d'un objet ?

Supposons que l'on crée deux objets 'nombre_complexe1' et 'nombre_complexe2' depuis la classe 'nombre_complexe'. On utilise le '.' Pour accéder à leurs attributs et/ou méthodes.

Exemples :

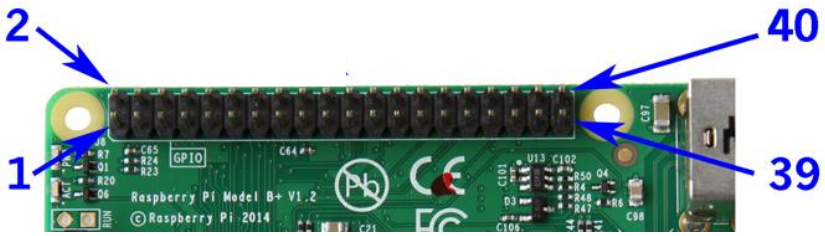
- nombre_complexe1.partie_reelle = 5
- nombre_complexe1.partie_imaginaire = 9
- resultat = nombre_complexe1.module()

Maintenant que ces notions de base ont été posées, on va pouvoir y regarder de plus près à travers l'écriture de plusieurs programmes. Mais comme l'on va utiliser une carte d'extension, il convient de parler un peu du connecteur GPIO de 40 broches de la Raspberry auparavant.

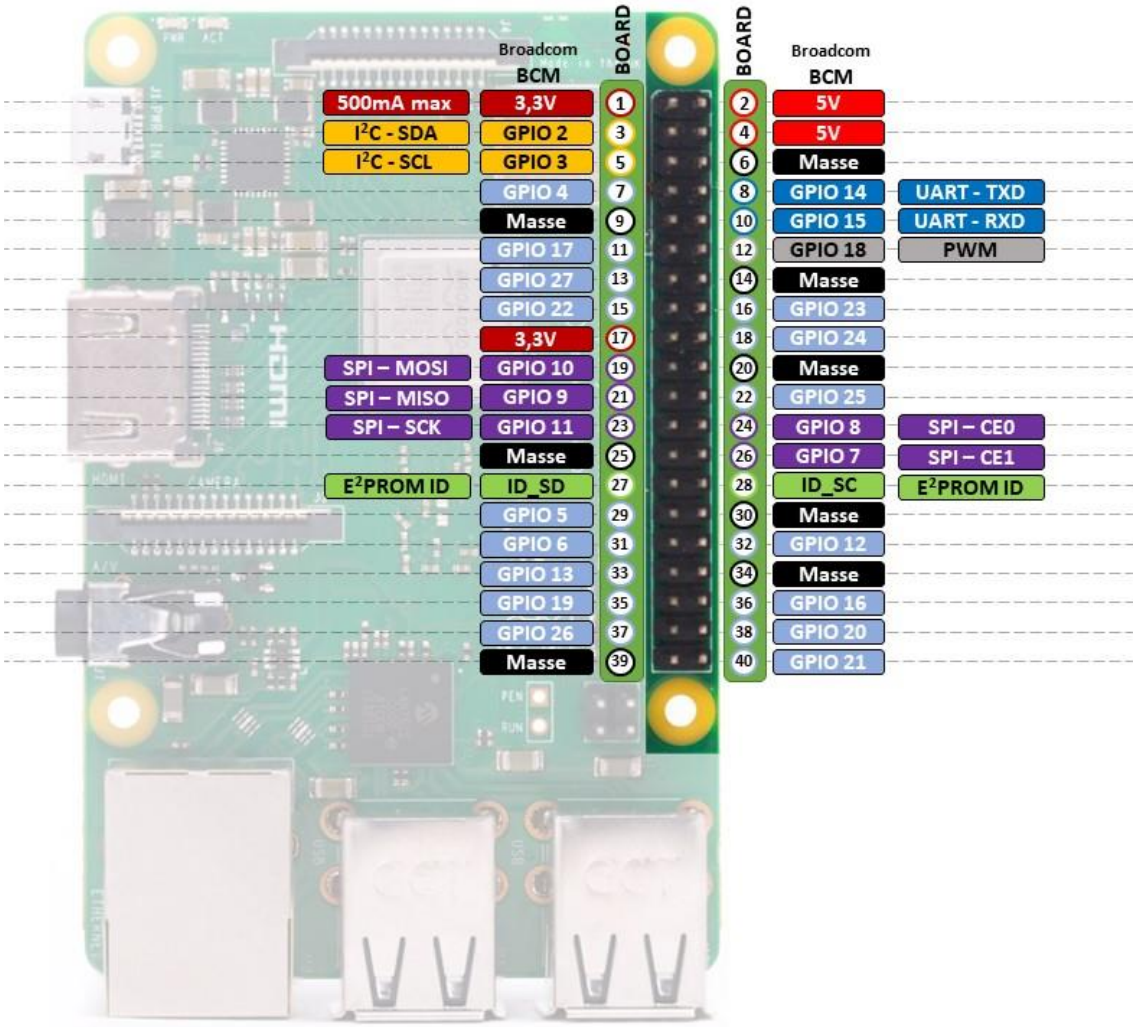
Présentation du connecteur GPIO

La carte Raspberry Pi (modèle B+) dispose d'un connecteur mâle de deux rangées de 20 broches appelé communément 'port GPIO' (General Purpose Input Output) :

Ces broches sont numérotées de la manière suivante :



Bien évidemment, toutes ces broches n'ont pas la même fonction. Le graphique ci-dessous vous donne leurs dénominations :



On voit donc bien qu'il n'y a pas que des broches d'entrée-sortie :

- 2 broches d'alimentation **3,3V** : ce sont des sorties permettant de fournir une tension régulée de 3,3V vers l'extérieur, mais attention, le courant est limité à 500mA.
- 2 broches d'alimentation **5V** : ce sont également des sorties mais sous 5V. Elles sont directement reliées au connecteur micro-USB de la Raspberry utilisé pour son alimentation. La Raspberry consommant au maximum 600-700mA, si vous utilisez un bloc secteur 5V/2A, vous disposerez de 1300 à 1400mA sur ces deux broches.
- 8 broches de **masse** (Ground).
- 2 broches **ID_SD** (27) et **ID_SC** (28) réservées uniquement à la communication avec une EEPROM I²C (type 24Cxx) installée sur une carte d'extension et qui contiendrait la configuration des I/O propres à cette carte.
- Il reste donc 26 broches I/O. Toutes ces broches peuvent être utilisées indifféremment en entrée numérique ou en sortie numérique. Certaines d'entre elles peuvent être configurées avec un usage particulier :
    - Les 2 broches n°8 et 10 peuvent être configurées en liaison série, respectivement en **TX** et **RX**.
    - Les 2 broches n°3 et 5 peuvent être configurées en liaison I²C, respectivement **SDA** et **SCL**.
    - Les 5 broches 19, 21, 23, 24 et 26 peuvent être configurées en liaison SPI (**SPI0**), respectivement **MOSI**, **MISO**, **SCLK**, **CE0** et **CE1**, permettant ainsi de piloter deux esclaves SPI sur un même bus SPI.

- Il y a une seconde interface SPI (**SPI1**) mais qui nécessite un peu de configuration pour pouvoir être utilisée.

Chaque broche I/O dispose de deux numéros pour l'identifier : un numéro correspondant à la broche sur le connecteur (**BOARD**) et un numéro correspondant au numéro de broche du microprocesseur embarqué (**BCM**).

Exemple : **GPIO9** → broche '**BOARD**' = **21** / broche '**BCM**' = **9**

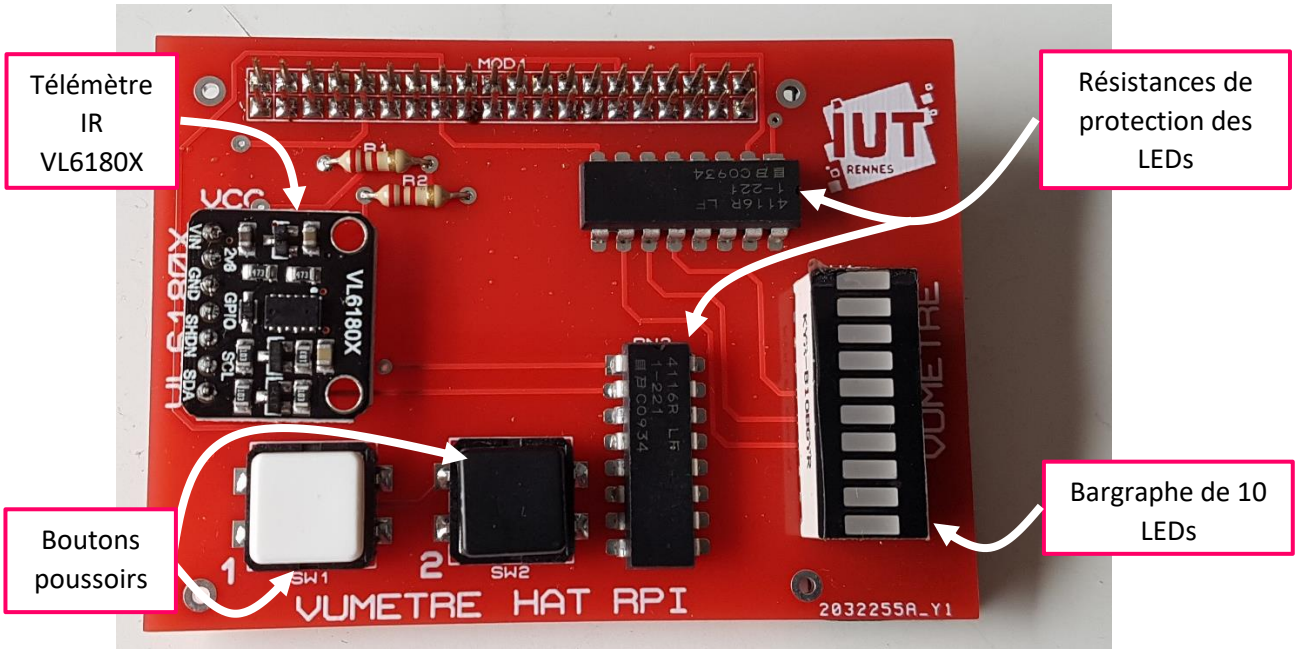
Il est important aussi d'avoir à l'esprit que la carte Raspberry Pi ne dispose pas d'entrées analogiques (ce qui est bien regrettable…) !! Il existe bien entendu sur le marché des convertisseurs analogique-numérique pilotables par une liaison I^2C ou par une liaison SPI.

REMARQUE TRES IMPORTANTE : ***Les broches GPIO sont prévues pour une utilisation avec des niveaux de tension 0V-3,3V. Elles ne sont pas protégées !! Appliquer des tensions plus élevées détériorerait de manière irréversible le port de la Raspberry !!***

Il n'y a pas non plus de protection contre les surintensités (économie oblige !!). Or au reset, les broches d'entrée-sortie sont configurées pour délivrer un courant max de 8mA. Conclusion : sous 3,3V, une résistance de charge inférieure à 400Ω créerait une surintensité qui entrainerait une détérioration du port GPIO !!

Découverte de la carte d'extension que l'on va utiliser :

Nous allons utiliser durant cette séance une carte d'extension conçue à l'IUT :

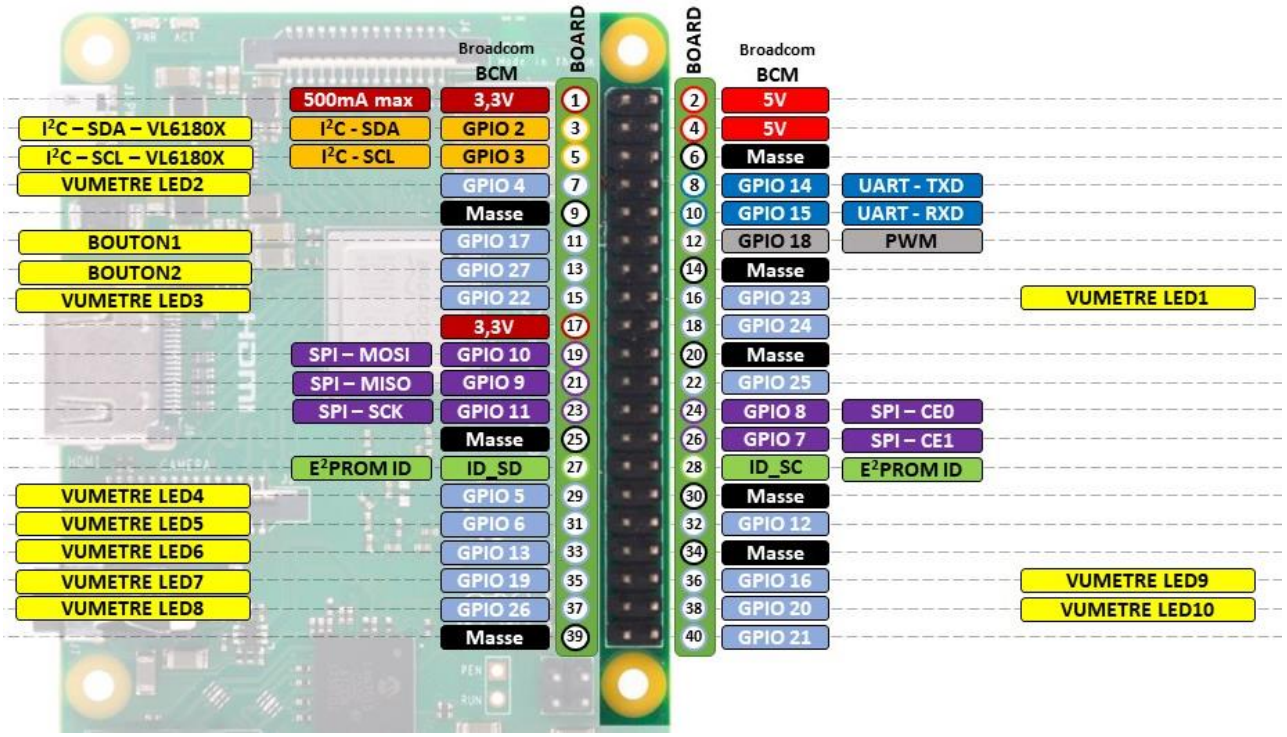


Elle se compose essentiellement :

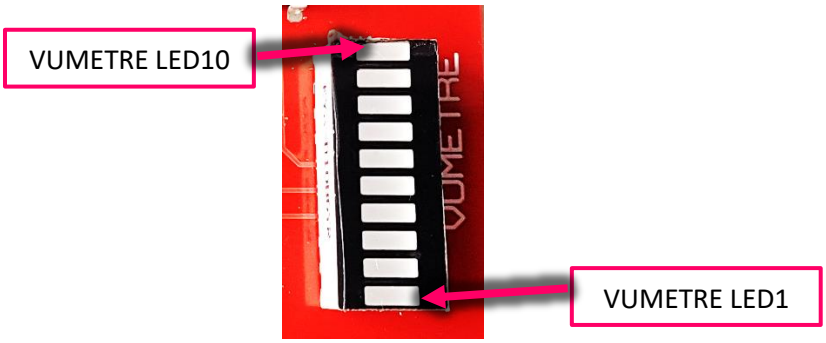
- D'un bargraphe de 10 LEDs (que nous utiliserons en vumètre par la suite) associé à des réseaux de résistances visant à protéger ces LEDs.

- D'un capteur de distance Infrarouge TOF de référence VL6180X qui utilise le bus I²C pour communiquer avec la Raspberry.
- De deux boutons poussoirs

Lorsque l'on va commencer à utiliser tout ces éléments, il faudra dans la programmation indiquer sur quelle(s) broche(s) ils sont connectés. Il est donc fondamental d'avoir un schéma de câblage. J'aurais pu vous fournir le schéma structurel sous Proteus, mais je me suis contenté ici d'un schéma de brochage qui est bien suffisant dans notre cas :



Pour le bargraphe, voici la répartition physique des LEDs :



Et pour les boutons poussoirs, ils sont câblés à la masse !!

Voilà, nous allons pouvoir (enfin) démarrer nos exercices de programmation.

EXERCICE 1 : 'Ex01.py'

On va au cours des prochains exercices utiliser des LEDs et des boutons poussoirs afin de découvrir quelques exemples simples d'utilisation du connecteur GPIO.

Librairie utilisée pour la gestion du port GPIO :

Pour cela, on va également utiliser une librairie Python permettant la gestion de ces ports d'entrée-sortie : **GPIO zero**
Le but ici est de découvrir quelques fonctionnalités de cette librairie.

Vous pourrez utiliser le document '**Memento**' mis à votre disposition en salle (à partir de la page 14) ou au format numérique dans le dossier '**IPE_IUT_TP2/Documentation**'. Vous trouverez également dans ce dossier le PDF de la documentation complète de cette librairie (également disponible en ligne à l'adresse suivante : https://gpiozero.readthedocs.io/en/stable/)

*Mais vous pouvez aussi vous aider de pages web avec des exemples dont il vous suffira de vous inspirer pour écrire votre propre code. ATTENTION toutefois… Lors de votre recherche, Vous allez certainement tomber sur des pages utilisant la librairie '**RPi.GPIO**'.*

*Certes, elle a été très utilisée et ce ne serait pas dramatique de l'utiliser encore aujourd'hui. Mais elle est plutôt ancienne et globalement obsolète (plus de mise à jour). A sa place est donc apparue la librairie '**GPIO zero**' qui est une classe pour la gestion du port GPIO de niveau beaucoup plus élevé. Veillez donc à faire vos recherches web en précisant 'raspberry python gpio zero' !!*

Ces librairies sont normalement déjà installées avec l'OS, mais il est toujours bon de les tenir à jour au cas où. Commencez donc par taper la commande suivante dans un terminal :

```
$ sudo apt install python3-gpiozero
```

Objectif de l'exercice :

Le but de ce premier exercice est faire clignoter une LED. Choisissez une LED du bargraphe et faîtes la clignoter au rythme de 0,5Hz.
Voici à quoi doit ressembler la structure de votre programme :

```
# Importation des librairies utiles
# ---------------------------------
from gpiozero import LED
from time import sleep

# Déclaration des objets
# ----------------------
# ECRIRE ICI LA DECLARATION D'UN OBJET LED_VUMETRE

# Boucle infinie
# --------------
while True :
    try :
        # ALLUMER LA LED
        # ATTENDRE UNE SECONDE
        # ETEINDRE LA LED
        # ATTENDRE UNE SECONDE
    except KeyboardInterrupt :
        # LIBERER LA BROCHE DE LA LED
        break
```

Quelques informations :

Pour écrire ces quelques lignes, il vous faudra trouver les informations correspondantes dans les documentations fournies ou sur la page web dédiée à cette librairie (utilisation de la classe '**LED**').

Remarque : le '**try… except**' permet ici de gérer correctement l'arrêt du programme. Effectivement, lorsque l'on quitte brutalement le programme (CTRL+C), la LED peut être dans sa phase allumée. Elle resterait donc allumée après la fin de l'exécution du programme ce qui n'est pas souhaitable. On gère donc l'exception '**KeyboardInterrupt**' dans laquelle on va libérer la broche de la LED et donc l'éteindre.

EXERCICE 2 : 'Ex02.py'

Objectif de l'exercice :

On va maintenant ajouter la gestion d'un bouton poussoir. Choisissez un des deux boutons poussoirs de la carte.
Vous devez faire en sorte que la LED précédemment utilisée change d'état à chaque appui sur le bouton poussoir

Quelques informations :

- Utilisez les ressources disponibles (mémento/documentation gpio zero/ web) pour compléter le programme précédent avec la LED en y intégrant maintenant la gestion par le bouton poussoir via la classe '**Button**'.
- Attention toutefois, le bouton étant relié à la masse, il est actif sur un niveau '0' et il lui faut une résistance de pull-up pour qu'il fournisse un niveau '1' au repos.

Faîtes contrôler votre programme par un enseignant.

EXERCICE 3 : 'Ex03.py'

Objectif de l'exercice :

Dans cet exercice, on va étendre le bargraphe à l'utilisation de toutes ses LEDs. Le but est de demander à l'utilisateur d'entrer une valeur de niveau entre 0 et 10 dans le terminal. Le nombre de LEDs allumées correspond au niveau choisi par l'utilisateur.

Quelques informations :

- Toutes les LEDs doivent être éteintes au début du programme
- On a ensuite une boucle infinie :
  - On demande à l'utilisateur un nombre '**niveau**' entre 0 et 10 dans le terminal (Cf. ex2 TP1)
  - On teste si la valeur entrée est entière via une exception (Cf. ex3 TP1)
  - Par de simples tests, on impose que le nombre soit bien compris entre 0 et 10
  - On éteint toutes les LEDs
  - On allume que les LEDs correspondant au niveau choisi par l'utilisateur

- On gère l'interruption du programme par le clavier en faisant en sorte d'éteindre toutes les LEDs avant de sortir définitivement du programme

Faîtes contrôler votre programme par un enseignant.

EXERCICE 4 : 'Ex04.py'

Objectif de l'exercice :

Dans son fonctionnement, cet exercice fera exactement la même chose que l'exercice précédent, mais sa structure s'appuiera maintenant sur une classe '**Vumetre**' que l'on va créer.

La classe 'Vumetre' :

En reprenant le formalisme présenté au début de ce document, voici à quoi va ressembler notre classe :

Vumetre
Pas d'attribut
eteindre() afficher(niveau) lire() inverser()

Cette classe 'Vumetre', pour rester sur un exemple simple, ne comporte que des méthodes :

- La méthode 'eteindre()' : elle permet d'éteindre toutes les LEDs simultanément
- La méthode 'afficher(niveau)' : elle permet d'allumer le nombre de LEDs correspondant au paramètre transmis 'niveau'
- La méthode 'lire()' : elle permet de récupérer le niveau correspondant au nombre de LEDs allumées
- La méthode 'inverser()' : elle permet d'inverser l'affichage du bargraphe (les LEDs allumées s'éteignent et inversement)

Commencer par créer un nouveau fichier '**Vumetre.py**' (situé dans le même dossier que les exercices de ce TP)
Il aurait été possible de vous fournir directement ce fichier, mais la notion de classe n'étant pas évidente, la seule façon de la découvrir est de tout écrire 'from scratch' !!

Pour vous aider, allez consulter les pages web suivantes :

- https://python-django.dev/page-apprendre-programmation-orientee-objet-poo-classes-python-cours-debutants
- http://fsincere.free.fr/isn/python/cours_python_classe.php
- http://sametmax.com/le-guide-ultime-et-definitif-sur-la-programmation-orientee-objet-en-python-a-lusage-des-debutants-qui-sont-rassures-par-les-textes-detailles-qui-prennent-le-temps-de-tout-expliquer-partie-1/
- https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/232721-apprehendez-les-classes

Voici donc à quoi doit ressembler votre fichier '**Vumete.py**' dans sa structure :

```
from gpiozero import LED

class Vumetre :
    """ Classe gérant un vumètre à LEDs
    (utilise la librairie GPIO ZERO 1.5)"""

    def __init__(self) :
        self.broches = [20, 16, 26, 19, 13, 6, 5, 22, 4, 23]
        self.leds = [LED(self.broches[0]),LED(self.broches[1]),LED(self.broches[2]),
                LED(self.broches[3]),LED(self.broches[4]),LED(self.broches[5]),
                LED(self.broches[6]),LED(self.broches[7]),LED(self.broches[8]),
                LED(self.broches[9])]

    def eteindre(self) :
        # Extinction de toutes les LEDs

    def afficher(self,niveau=0) :
        # Extinction de toutes les LEDs d'abord
        # Allume ensuite uniquement les LEDs concernées par la valeur de 'niveau'

    def lire(self) :
        niveau = 0
        # Recupère la valeur de la dernière LED allumée
        return niveau

    def inverser(self) :
        # changement d'état ('toggle') de chacune des LEDs
```

Quelques informations pour l'écriture du code 'Ex04.py' :

- Il reprend globalement la structure du programme 'Ex03.py'
- Pour utiliser votre classe 'Vumetre', il faut déjà l'importer au début de votre programme :
  - **from Vumetre import Vumetre**
- Vous devez ensuite créer un objet ('instancier') basé sur cette classe :
  - **Bargraphe = Vumetre()**
- Dans le reste de votre programme, vous accéderez aux différentes méthodes de votre objet par : '**Bargraphe.methode()**'

Faîtes contrôler votre programme par un enseignant.

Remarque : Si vous ne parvenez pas à écrire cette classe malgré les informations de ce document et celles fournies par vos enseignants, vous pouvez toujours passer cet exercice et faire en sorte de conserver la structure de l'exercice 3 pour les prochains exercices. Mais essayez tout de même de fournir les efforts nécessaires pour y parvenir.

TUTORIEL N°1

Objectif de ce tutoriel :

On va mettre en œuvre la démarche visant à installer et à utiliser une librairie Python trouvée sur le web permettant d'exploiter un composant spécifique rapidement sans avoir à écrire du code depuis zéro !!

On va ici s'intéresser au dernier composant disponible sur la carte d'extension : le capteur **VL6180X**.
Il s'agit d'un télémètre INFRAROUGE ToF.

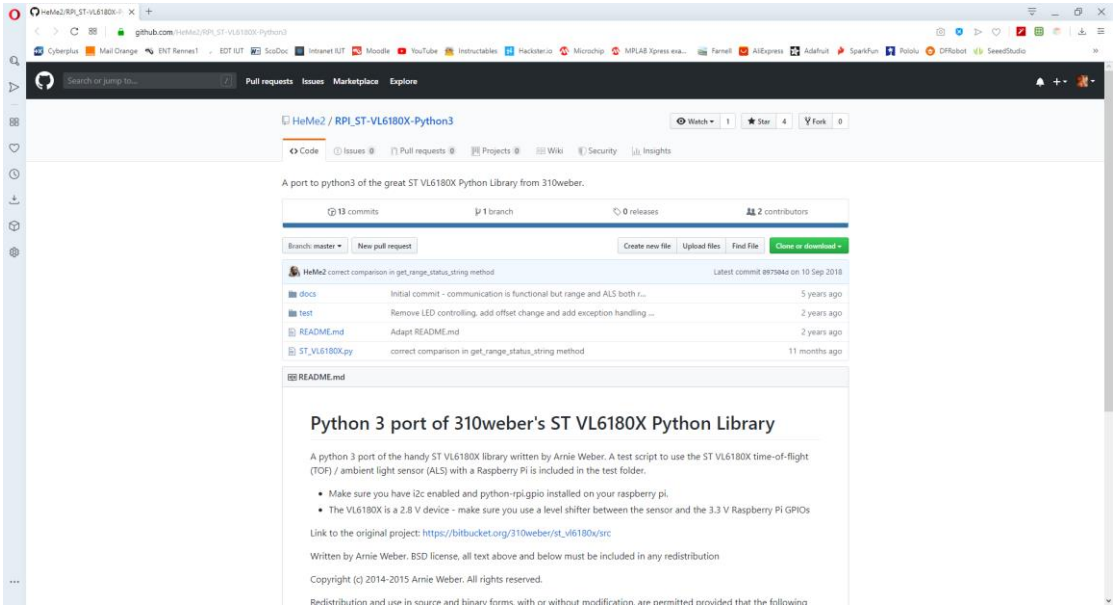
Quelques informations générales sont disponibles à l'adresse suivante :
https://www.st.com/en/imaging-and-photonics-solutions/vl6180x.html
Repérez notamment quelle est la plage de mesure utile et fiable de ce capteur.
On va dans cet exercice voir comment récupérer une librairie pour ce capteur et l'utiliser.

Récupérer une librairie Python 3 pour le capteur VL6180X :

Essayez simplement de taper dans un moteur de recherche les mots clefs '**python**' et '**VL6180X**'.
Vous allez voir qu'il existe de nombreux sites ayant ces deux mots dans leur contenu.
Seulement, certains sont purement commerciaux, d'autres font référence à '**micropython**' ou '**circuitpython**' et non '**python**'.
En ajoutant le mot clef '**Raspberry**', on élimine quelques pages inutiles, mais le choix est encore vaste !!
En ajoutant le mot clef '**Github**', on est sûr de tomber sur du code, ce qui nous intéresse davantage.
Mais le choix reste large. Quelle solution alors ?
Eh bien, il faut passer du temps à étudier les codes proposés, éliminer ceux qui sont en Python 2.x, ou ceux qui ne sont pas assez documentés, ou ceux qui sont trop limitatifs vis-à-vis des performances des composants ciblés, …
Bref, cela peut être rapide comme cela peut être fastidieux !!

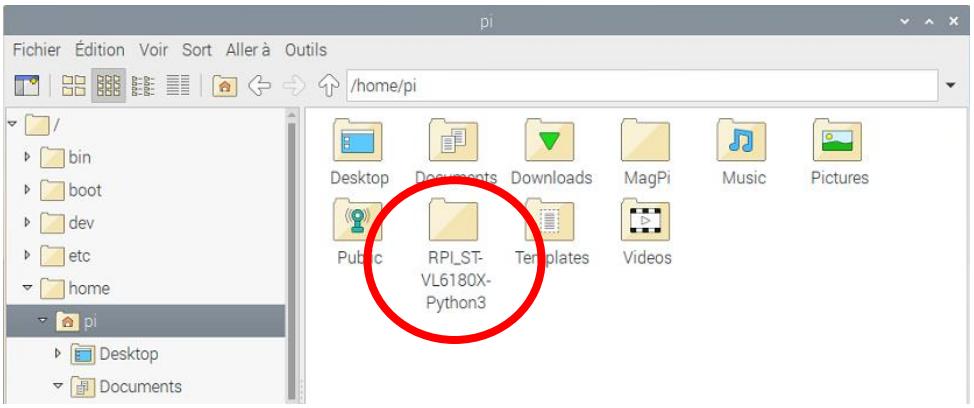
Comme il s'agit d'une séance de TP, je vais vous simplifier la tâche en vous fournissant l'adresse d'une librairie qui fonctionne correctement sous Python 3.x :
https://github.com/HeMe2/RPI_ST-VL6180X-Python3

Consultez donc cette page :

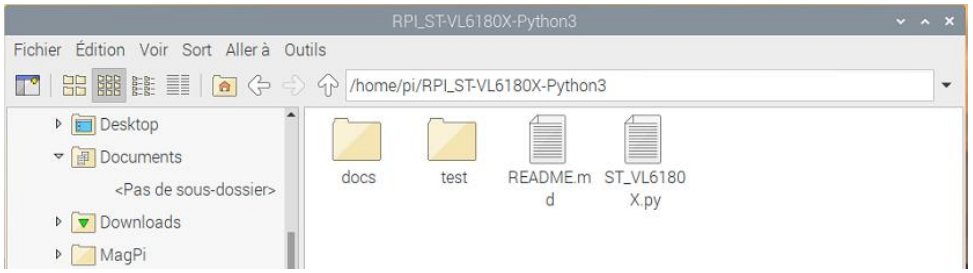


Je vais maintenant vous indiquer la démarche à effectuer pour installer une librairie externe. Cette démarche vous servira en projet et/ou en stage si votre sujet se fait sur Raspberry.

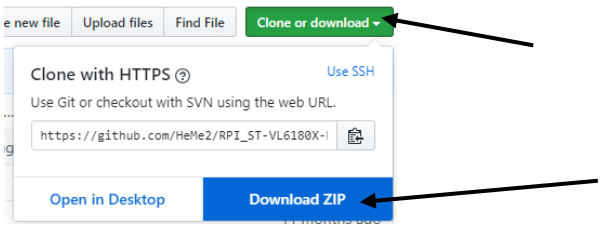
1. Commencer par ouvrir un terminal (vérifiez bien que vous êtes sous la racine de votre répertoire 'utilisateur' → pour cela, aucun nom de dossier ne doit figurer en 'bleu' devant le prompt '$')
2. On va cloner l'arborescence de la page Github par la commande suivante :
   $ git clone https://github.com/HeMe2/RPI_ST-VL6180X-Python3
3. Une fois le clonage terminé, en ouvrant un explorateur de fichiers, vous devriez voir dans le dossier 'pi' le dossier de la librairie téléchargé :



4. En allant dans ce dossier, vous y retrouverez la même arborescence que sur la page web 'Github' de la librairie :

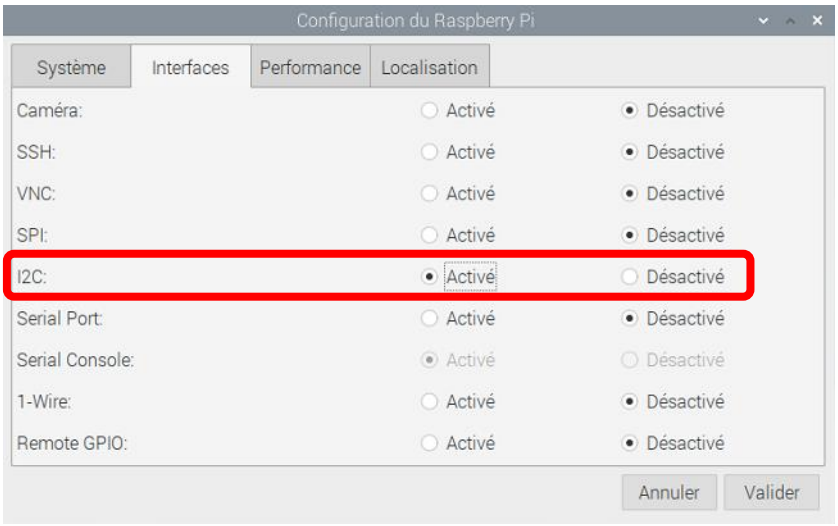


5. A partir de là, deux suites sont possibles :
   a. Cette arborescence ne contient aucun fichier 'setup.py' : il suffit alors de copier dans votre répertoire de travail les fichiers Python disponibles à ce niveau. Aucune installation n'est nécessaire mais il faudra toujours disposer des fichiers de la librairie dans chaque répertoire de travail. On se retrouve dans le cas de notre fichier 'Vumetre.py'.
   b. Cette arborescence contient un fichier 'setup.py' : la librairie nécessite alors une installation par le biais de la commande : **$ sudo python3 setup.py install**
   Une fois l'installation faite, on peut importer la librairie dans n'importe quel fichier Python dans n'importe quel dossier, elle sera toujours visible.
6. *Une remarque supplémentaire :* si pour une raison quelconque, vous ne pouvez pas cloner le dossier Github, vous pouvez toujours le télécharger via le bouton ci-dessous. Vous pourrez alors le dézipper dans le dossier voulu et arriver à l'étape n°5 de la démarche.



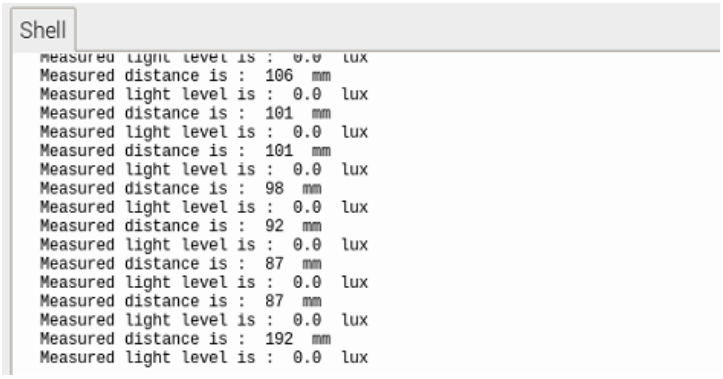
Si la librairie est bien documentée, elle est généralement accompagnée d'exemples pour la tester. C'est ce que nous allons faire dans la suite de ce tutoriel.

Mais tout d'abord, on a évoqué quelque chose de très important : ce composant VL6180X utilise le protocole I2C pour communiquer avec la Raspberry.
Or par défaut, il n'est pas activé. Il ne faut donc pas oublier d'aller les 'préférences > Configuration du Raspberry Pi' et de le valider dans l'onglet 'interfaces' :



Rendez vous maintenant dans le dossier 'test' de la librairie clonée et ouvrez le fichier disponible dans l'éditeur 'Thonny'. Puis lancez le code. Vous devriez obtenir le message suivant :

Essayez de comprendre l'origine de cette erreur et corrigez-la afin que ce programme de test puisse finalement fonctionner !
Vous devriez obtenir des résultats de ce type en modifiant la position d'un obstacle au-dessus du capteur :



Ne passez pas à la suite tant que vous ne les avez pas obtenus.

EXERCICE 5 : 'Ex05.py'

Objectif de l'exercice :

On va combiner dans cette exercice la mise en œuvre du vumètre de l'exercice 4 avec l'utilisation du capteur de distance VL6180X. Le but est d'allumer une LED sur le vumètre par centimètre. Si l'obstacle est à 6cm, on allumera ainsi 6 LEDs. On se limite donc à une distance de détection de 10cm.

Si on restreint le programme de test du tutoriel à l'essentiel, voici ce que l'on pourrait écrire :

```
1  # Importation des librairies utiles
2  from time import sleep
3  from ST_VL6180X import VL6180X
4
5  # Instanciation de l'objet 'telemetre' avec son adresse I2C
6  telemetre_adresse = 0x29
7  telemetre = VL6180X(address=telemetre_adresse, debug=False)
8
9  # Essai de communication avec le capteur
10 telemetre.get_identification()
11 if telemetre.idModel != 0xB4:
12     print("Capteur non valide id: {}".format(telemetre.idModel))
13 else:
14     print("Modèle du capteur: {}".format(telemetre.idModel))
15     telemetre.default_settings()
16
17     # Boucle infinie de mesure da la distance
18     while True :
19         distance = telemetre.get_distance()
20         if distance > 0 :
21             print("%d mm, %d cm" % (distance, (distance/10)))
22         sleep(1)
```

En vous appuyant sur votre travail de l'exercice n°4 (ou du n°3 pour ceux/celles qui n'ont pas réussi à écrire la classe 'Vumètre'), retirez le code inutile et complétez-le avec celui du capteur pour répondre au besoin de l'exercice.

Faîtes contrôler votre programme par un enseignant.

EXERCICE 6 : 'Ex06.py'

Objectif de l'exercice :

On va compléter l'exercice précédent en ajoutant à l'écran une fenêtre qui nous affiche la distance mesurée en cm.

Comment créer une fenêtre graphique à l'écran ?

Il n'y a pas dans la base du langage Python de méthodes et d'attributs permettant de gérer des interfaces graphiques (**GUI : Graphical User Interface**).
On va donc là aussi avoir recours à une librairie extérieure. Ok, mais laquelle ?

Il y en a un grand nombre. Voici quelques exemples :

- Tkinter
- GTK
- Qt
- Pygame
- PySide
- WxPython
- (…)

Le choix n'est pas simple encore une fois et une fois choisi, la prise en main d'une telle librairie demanderait beaucoup de temps.
Je vais donc vous proposer une solution alternative : GuiZero
On l'utilisera à plusieurs reprises au cours de ces séances sur Raspberry.
Cette librairie est basée sur Tkinter mais en version simplifiée. Toutes deux sont préinstallés dans l'OS Raspbian.
Toutefois, pour être sûr d'avoir la dernière version corrigée d'éventuels bugs, il convient de la mettre à jour via un terminal :
$ sudo pip3 install guizero
Soit une mise à jour s'effectuera, soit le terminal vous informera que la version installée est la plus récente.

Vous trouverez toute la documentation de cette librairie à l'adresse suivante :
https://lawsie.github.io/guizero/about/

Comme nous avons besoin d'une simple zone de texte, voici un exemple que vous pouvez tester qui affiche du texte (https://lawsie.github.io/guizero/text/) :

```
1  # Importation des librairies utiles
2  from guizero import App, Text
3
4  # Création de la fenêtre d'affichage et de l'objet 'text'
5  fenetre = App(title="simple test", height=100, width=350, bg="#FFFFFF")
6  texte_a_afficher = Text(fenetre, text="????", color="#FF0000")
7  texte_a_afficher.text_size=55
8
9  # Affichage
10 fenetre.display()
```

En observant le résultat, vous devriez rapidement comprendre le rôle de chacune des lignes. Modifiez quelques paramètres pour vérifier votre interprétation.
Pour la couleur, il s'agit d'un code hexa RVB au format '#RRVVBB'. Vous pouvez utiliser la page web suivante pour régler les couleurs souhaitées : https://htmlcolorcodes.com/fr/selecteur-de-couleur/

Le but maintenant va consister à créer la fenêtre suivante dans laquelle la valeur de la distance se met à jour continuellement (dans le même temps, le vumètre de LEDs continue également à afficher la distance) :

Si la mesure ne devait s'effectuer qu'une seule fois au lancement du programme, vous arriveriez assez rapidement au code attendu.
Mais ici, le rafraîchissement de l'affichage doit être continuel (par exemple toutes les 100ms).
Pour cela, je vous conseille fortement de lire et de vous inspirer de la page suivante (notamment la solution apportée) : https://lawsie.github.io/guizero/blocking/

Faîtes contrôler votre programme par un enseignant.

EXERCICE 7 : 'Ex07.py'

Objectif de l'exercice :

Rien à écrire ici, juste une simple démonstration de l'utilisation de ce capteur dans un cadre différent. On va l'utiliser ici comme organe de contrôle.

Vous avez juste à ouvrir le programme fourni et le lancer.