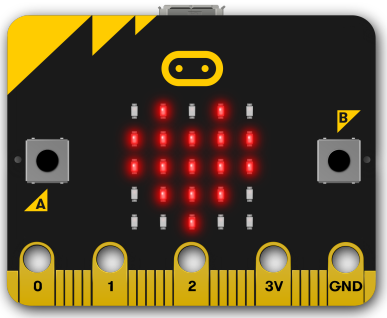
Vincent Le Mieux

Découvrir le langage Python en programmant un microcontrôleur !



00000001

python powered

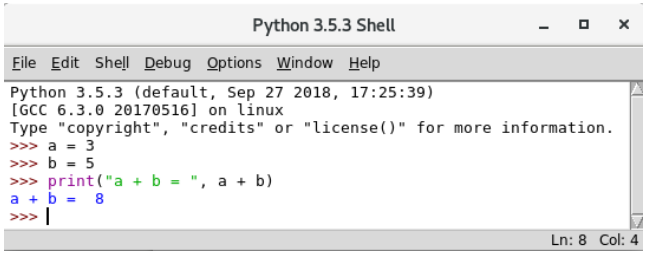
Table des matières

1 Introduction :

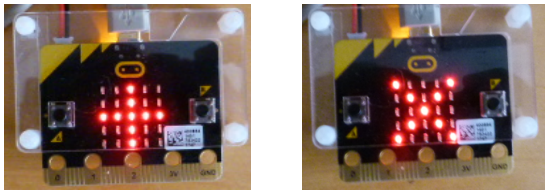
Que ce soit en Mathématiques, en Sciences Physiques ou dans les enseignements liés au Numérique et aux Sciences Informatiques, les programmes rédigés pour la réforme 2019 du lycée font apparaître clairement et à plusieurs reprises la nécessité d'introduire des éléments de programmation avec le langage **Python.** L'utilisation d'un **microcontrôleur ou d'objets connectés** est demandée dans ces programmes (sauf celui de mathématiques)

Cette nouveauté dans les programmes de seconde et de première se retrouvera très certainement aussi dans les programmes de Terminale.

La découverte du langage Python est très souvent réalisée autour de l'utilisation de la console qui reste très limitée au niveau IHM (Interface Homme Machine), aussi bien dans l'aspect visuel que dans les possibilités d'interactivité :

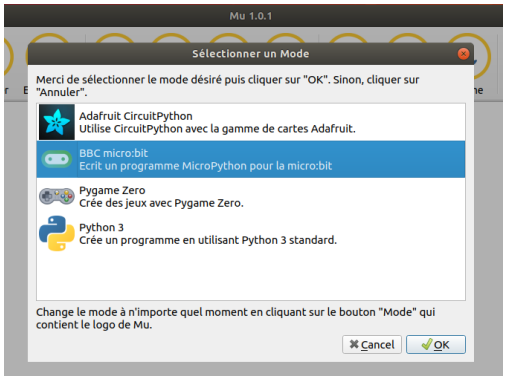


Nous vous proposons dans cet ouvrage une autre approche : découvrir le langage Python en programmant une carte microcontrôlée permettant par ses ressources intégrées (bouton poussoirs, afficheurs à Leds ...) une approche plus conviviale mais tout aussi rigoureuse de la programmation :

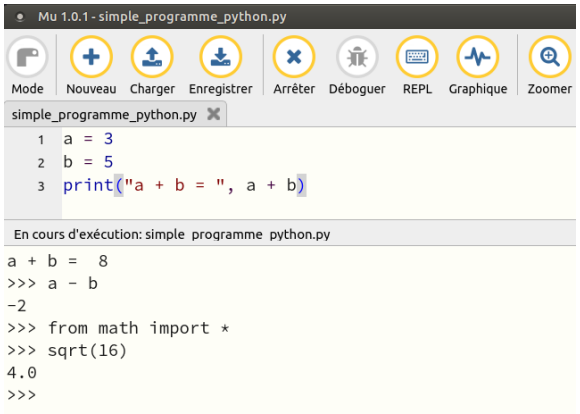


Un exemple : on peut découvrir la boucle « while » pour afficher ces symboles sur l'afficheur et aller plus loin avec la notion de fonction, voire même de liste associée à la boucle « for » !

Pour ne pas être enfermé dans un logiciel qui ne permettrait de faire que de la programmation sur de l'électronique embarquée, nous utiliserons un logiciel librement téléchargeable (mu-editor), fonctionnant sous Windows, Linux ou Mac, et qui peut être utilisé sous différents modes :



- Le mode BBC micro:bit sera utilisé pour programmer la carte microcontrôlée
- On pourrait aussi développer de petits jeux avec PyGame Zéro…
- Le mode Python 3 permettra de retrouver un environnement de développement Python classique… et sa console :



Typographie utilisée dans ce manuel :

Ce manuel a été intégralement composé avec le logiciel Writer de la suite bureautique LibreOffice. Les logiciels Gimp et Inkscape ont permis la gestion de certaines images. Merci à tous ces développeurs !

On y trouvera :

- du code écrit ainsi :

```
from microbit import *
i = 0
while i < 5 :
    display.set_pixel(i, 2, 9)    # x va varier de 0 à 4 ; y fixé à 2
    i = i + 1
```

- ***Des remarques écrites en violet avec une police mise en italique***
- **Des exercices dont les solutions sont fournies à la fin de ce manuel. On reconnaîtra les exercices à cette typographie (en gras et bleu).**

Dans les solutions, les parties de code à repérer tout particulièrement sont « fluotées » en jaune :

```
display.set_pixel(2,1,9)
display.set_pixel(2,2,9)
display.set_pixel(2,2,9)
display.set_pixel(3,2,9)
```

Pour aller plus loin, des remarques sont parfois ajoutées dans les corrections : il est important d'aller les voir.

Repérage : un exercice noté Exercice 4.3.2_2 signifie qu'il s'agit du deuxième exercice du paragraphe 4.3.2. Le corrigé reprend la même numérotation. Si un programme est associé, il portera alors le nom Ex_432_2.py. Il est conseillé d'utiliser la même notation pour pouvoir s'y retrouver plus facilement

2 Le microcontrôleur

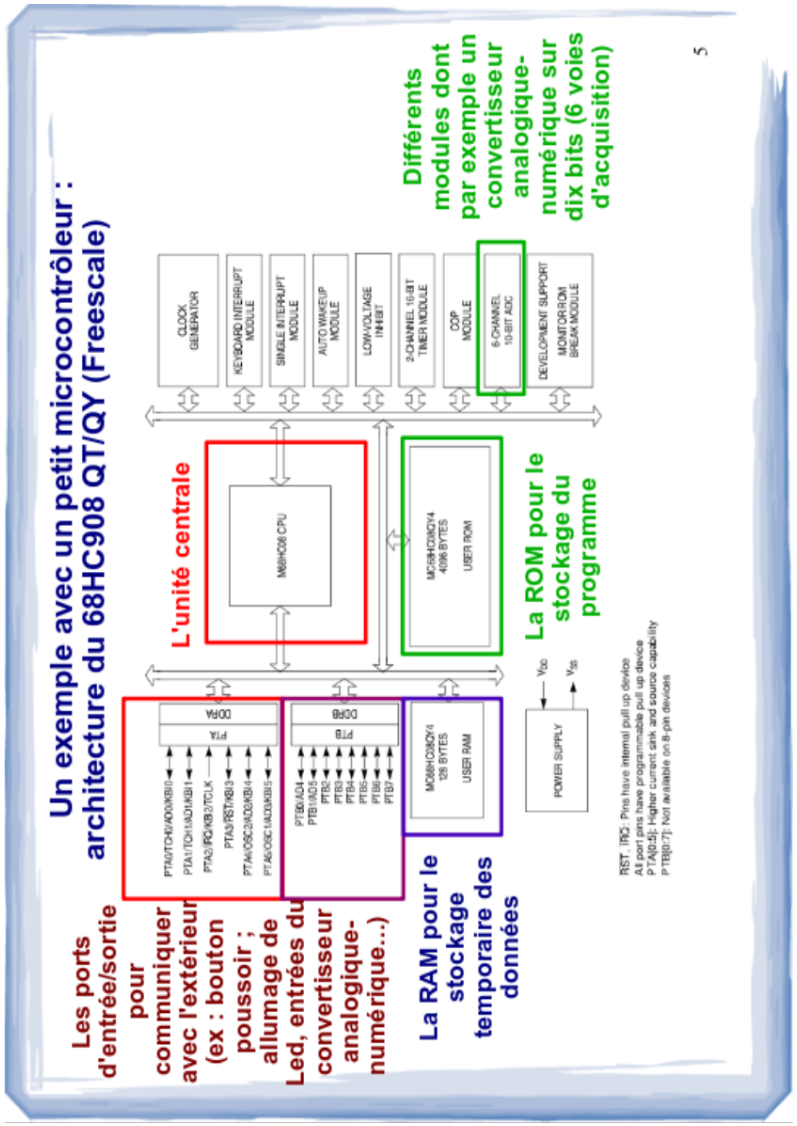
2.1 Présentation

Si le terme de microcontrôleur est bien moins connu que celui de microprocesseur, ce composant est pourtant omniprésent dans notre environnement :

Pour se faire une idée de ce qu'est un microcontrôleur, on peut simplement repenser à la constitution d'un ordinateur. On y trouve :

- un microprocesseur (Intel, AMD)
- de la mémoire vive (RAM) pour du stockage temporaire (quelques Go « giga octets = 10^9 octets)
- de la mémoire pour du stockage permanent (disque dur : plusieurs centaines de Go)
- des ports de communication pour dialoguer avec l'extérieur (USB, Ethernet, WiFi, carte son, carte graphique…)

Le microcontrôleur, c'est un peu tout cela dans un format réduit (de l'ordre du cm^2) avec quelques nuances cependant !



Le schéma ci-contre représente sous forme de blocs fonctionnels le contenu d'un ancien « tout petit » microcontrôleur (dont une version réduite tient dans un boîtier de 8 broches seulement...)

On y trouve :

- un microprocesseur (CPU : Central Processor Unit)
- de la mémoire de type RAM (ici 128 octets seulement)
- de la mémoire non volatile pour stocker le programme (c'est l'équivalent du disque dur pour un ordinateur). Cette mémoire est maintenant reprogrammable ce qui permet de mettre à jour le programme ("firmware") (ici 4096 octets = 4 ko … !!!)
- des ports d'entrées / sorties pour communiquer avec l'extérieur
- un convertisseur analogique-numériques (CAN) pour faire de la mesure.

Avec un tel composant, on pouvait envisager la création par exemple d'un thermostat d'ambiance simple :

- on connecte sur les ports d'entrée / sortie un écran, deux boutons poussoirs, un capteur de température et un relais pour piloter la chaudière
- Les deux boutons poussoirs permettent de choisir une température de consigne, lisible sur l'écran.
- un capteur de température, relié au convertisseur analogique numérique donne au microprocesseur la valeur de la température ambiante. Celui-ci la compare à la valeur choisie par l'utilisateur et déclenche ou arrête en conséquence la chaudière par une action électrique sur le relais.

Mais, ce même composant peut aussi être affecté à des tâches très différentes, c'est juste une question de programmation. De plus, le fabricant propose souvent différents modèles de microcontrôleurs d'une même famille : selon la complexité de l'application à réaliser, on peut choisit alors le plus adapté (pour avoir plus d'entrées/sorties, un module de communication USB etc)

2.2 Langage de programmation

Initialement programmés en assembleur (langage spécifique à la famille à laquelle appartient le microcontrôleur), ces composants sont le plus souvent maintenant programmés dans un langage universel de plus haut niveau, plus simple donc, mais plus gourmand en ressources, un problème qui n'en est plus un depuis l'extraordinaire montée en puissance que ces composants ont subi en quelques années.

Le langage de haut niveau le plus utilisé sur microcontrôleur, c'est le C. Rappelons qu'un programme rédigé en C sera compilé avant d'être implanté dans la cible (le microcontrôleur) :

- le programme est d'abord rédigé avec un éditeur de texte, puis compilé avec un "cross-compilateur" pour être transformé en langage machine adapté au microcontrôleur choisi. La taille du programme obtenu à l'issue de la compilation est souvent supérieure à celle obtenue à ce que l'on aurait obtenu avec une programmation en langage d'assemblage, mais le gain en temps de développement (et en portabilité / réutilisation du code) est tel que le C s'est imposé depuis une vingtaine d'années comme le langage de programmation de prédilection sur microcontrôleur chez les professionnels.

Le petit exemple ci-dessous montre l'intérêt d'un langage de haut niveau. On souhaite faire une boucle vide qui tourne dix fois (pour faire une temporisation dans le déroulement du programme) :

Famille	HC08 (Motorola)	MCS-51 (Intel)
en assembleur	**debut: lda #!10** **boucl: deca** **bne boucl**	**debut: mov A, #10** **boucl: dec A** **jnz boucl**
en C	**unsigned char i ;** **for (i = 0 ; i < 10 ; i++)** **{** **}**	

- en assembleur, le code est spécifique à chacune de ces deux familles
- en C c'est le même code : le compilateur créera le code adapté à chacun des deux microcontrôleurs. C'est cet aspect universel, (auquel s'ajoute le fait qu'au cours des études supérieures un ou

plusieurs langages de haut niveau auront été appris) qui incite à utiliser un tel langage

L'augmentation importante de la taille mémoire disponible sur les microcontrôleurs récents permet même aujourd'hui de programmer ces composants en Python.

Python quant à lui est un langage interprété qui nécessite donc un "interpréteur" qui doit être implanté dans le système (ici le microcontrôleur) Cet interpréteur exécutera ligne par ligne le code Python (ce qui permet éventuellement un travail dans une console). La mémoire "ROM" du microcontrôleur doit donc être suffisante pour contenir :

- l'interpréteur Python
- le programme Python que l'on crée pour réaliser une application donnée

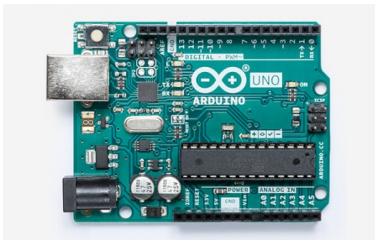
C'est une telle solution que l'on vous propose dans les pages suivantes : utiliser une carte microcontrôlée, de faible coût, programmable en Python, ce qui permettra de répondre à la demande des nouveaux programmes de cette réforme 2019 : utiliser un microcontrôleur et programmer en Python

Pour en savoir plus sur les microcontrôleurs, un diaporama développant davantage ce qui vient d'être dit, est disponible sur le site de l'auteur à la page :

http://laboiteaphysique.fr/site2/index.php/electronique/le-microcontroleur

2.3 Choix d'une carte microcontrôlée

Pour expérimenter avec un microcontrôleur, on utilise bien souvent maintenant des cartes de développement prêtes à l'emploi qui permettent la (re)programmation du microcontrôleur.



On peut citer le très connu système Arduino dont la carte de développement peut accueillir une multitude de cartes additionnelles différentes (écrans tactiles ou non, modules radio etc...) :

source : www.arduino.cc

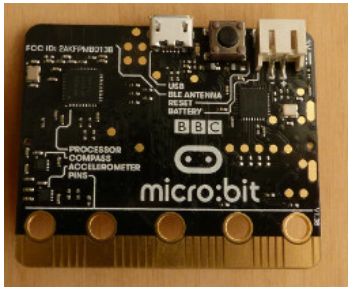
Mais Arduino se programme en C :

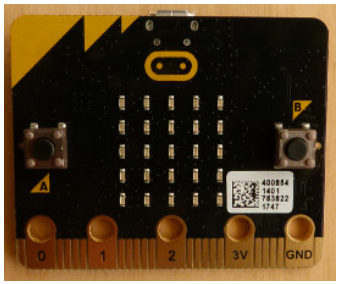
```
int led = 13;
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

(exemple de programme pour faire clignoter une Led)

… ce qui ne convient pas à notre cahier des charges !

Une alternative intéressante, et utilisée depuis plusieurs années Outre-Manche, est fournie par la BBC sous la forme d'une carte embarquant un microcontrôleur bien plus puissant que celui présent sur la carte Arduino, ce qui lui donne la **possibilité d'être programmé en Python**. Il s'agit de la carte BBC Micro:Bit





Face Arrière	Face avant
avec "toute" l'électronique	avec deux boutons poussoirs et un panneau de 5x5 Leds rouges

Le tableau suivant permet de faire une comparaison entre la très connue (mais un peu datée) carte Arduino et la moins connue (en France tout du moins) carte BBC Micro:Bit.

On a surligné en vert les avantages qu'a une carte par rapport à l'autre.

On voit au premier coup d'œil que la carte Micro:Bit est, de base, et pour un prix équivalent, plus puissante et bien mieux équipée que la carte Arduino Uno. C'est sur ces caractéristiques et sur la possibilité de la programmer en Python que nous avons décidé de nous y intéresser.

Comparaison de ces deux cartes de développement :

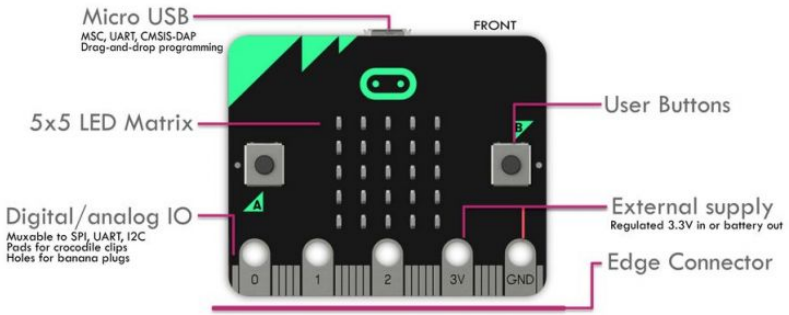
	ARDUINO UNO	BBC:MICROBIT
Microcontrôleur	ATmega328P	ARM Cortex M0
Architecture	8 bits	**32 bits**
Fréquence	16 MHz	16 MHz
Mémoire Flash	32 KB	**256 KB**
RAM	2 KB	**16 KB**
EEPROM	1 KB	
Alimentation	**5 V**	3,3 V
Entrées/Sorties	14	19
CAN	10 bits	10 bits
Bus série	I2C, SPI,UART,USB	I2C, SPI,UART,USB
Capteurs/Actuateurs		
Boutons poussoirs	-	**2**
LEDs	1	**25 (matrice 5x5)**
Accéléromètre	-	**1**
Boussole	-	**1**
Capteur de lumière	-	**(avec les Leds)**
Capteur de T°	-	**T° du chip**
Antenne radio	-	**1**
Programmation	C	**Python**
Documentation	**Importante**	Faible
Cartes additionnelles	**Très variées**	Beaucoup moins
Prix	~ 20 €	~ 20 €

Remarque concernant la tension d'alimentation : la diminution de la tension d'alimentation nécessaire au fonctionnement de l'électronique numérique a été un souci constant chez les fabricants de puces. Une tension d'alimentation de 3,3 V présente l'intérêt de pouvoir alimenter le montage avec deux piles de type AA ou AAA ce qui est un gros avantage par rapport à une alimentation sous 5 V. Cela diminue effectivement le coût énergétique à l'utilisation, ainsi que l'encombrement et le poids des appareils. Il faut cependant que toute la chaîne électronique soit capable de fonctionner avec cette tension d'alimentation. Dans la liste des cartes ou composants additionnels connectables sur ces cartes de développement, il faudra veiller à la compatibilité au niveau tension. Or actuellement, du fait du fort développement du système Arduino (qui commence quelque peu à dater)..., on trouve, dans le domaine amateur, beaucoup de composants additionnels en 5 volts.

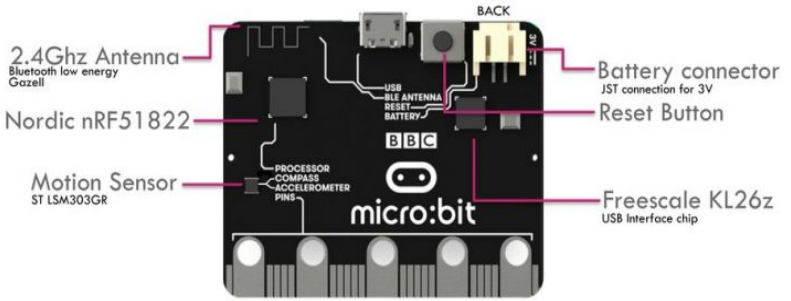
3 Démarrer avec la carte Micro:Bit

Les schémas ci-dessous résument de façon visuelle l'ensemble des ressources matérielles disponibles sur la carte BBC Micro:Bit

Face avant :



Face arrière :



Le microcontrôleur, c'est la puce carrée référencée Nordic nRF51822

3.1 Choix du matériel :

Du matériel Micro:Bit est en vente notamment chez les distributeurs suivants (cités par ordre alphabétique) : Conrad, Farnell, Gotronic, Kubii, Lextronic, Velleman

Pour démarrer, il faut au minimum : une carte Micro:Bit et un câble micro USB (qui permet d'alimenter la carte et de communiquer avec elle)

Si l'on souhaite pouvoir utiliser la carte de façon autonome (lorsqu'elle a été programmée) alors il y a nécessité de l'alimenter avec deux piles de 1,5V via le connecteur d'alimentation : il faut alors se procurer le coupleur de piles dédié.

Il pourrait être judicieux de protéger la carte Micro:Bit. Pour cela il existe des dispositifs en dur (acrylique) ou souples (caoutchouc).
Nous avons choisi de prendre un kit de démarrage contenant ces différents éléments (Référence VMM001). Voici le contenu de la boite :

- une carte BBC Micro:Bit
- un câble USB de bonne facture (contacts dorés) et de longueur 70 cm environ
- un coupleur de piles et 4 piles AAA (ce qui donne deux piles d'avance)
- un boîtier de protection en polycarbonate à monter (plaques protégées par un papier à retirer
- une "notice" (qui en gros renvoie sur le site Micro:Bit !)

Rque : il existe plusieurs "Starter kits" différemment fournis. Attention à la longueur du cordon USB fourni. Selon que le PC sera sur la paillasse ou en dessous, la longueur utile ne sera pas la même et pourrait nécessiter l'achat de rallonges USB

La carte Micro:Bit avec sa protection acrylique :

Rque : cette protection acrylique présente un avantage certain : celui de pouvoir voir tous les composants de la carte, mais présente deux inconvénients :

- le connecteur n'est pas protégé … normal il est fait pour être utilisé, mais pour un cours nécessitant juste la carte, on aurait intérêt à se tourner vers ce type de protection qui laisse néanmoins l'accès aux broches P0, P1 et P2 ainsi qu'à l'alimentation 3,3V et la masse

- elle est trop épaisse pour permettre à la carte d'être insérée dans la platine Grove que l'on décrira plus loin dans ce manuel.

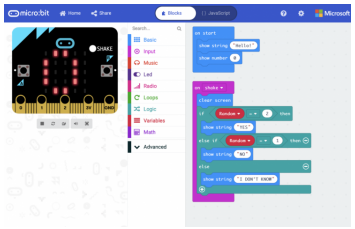
La carte Micro:Bit est livrée avec un programme de démonstration qui met en évidence les possibilités de l'afficheur de 25 Leds embarqué. Ce programme ne demande qu'à démarrer !

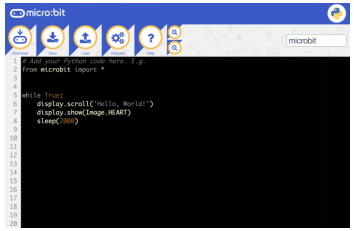
Il suffit pour cela d'alimenter la carte :

- soit à l'aide du coupleur de piles
- soit en connectant la carte sur un port USB à l'aide du câble mico USB

3.2 Installation du logiciel

Le site MicroBit propose deux logiciels en ligne pour programmer le microcontrôleur :





A gauche : l'éditeur "MakeCode" pour programmer de façon graphique (à la manière de Scratch pour ceux qui connaissent). Cet éditeur est plutôt destiné à de jeunes programmeurs.

A droite : un éditeur Python. (Inconvénient : il faut être sûr de disposer d'une connexion internet pour le faire fonctionner

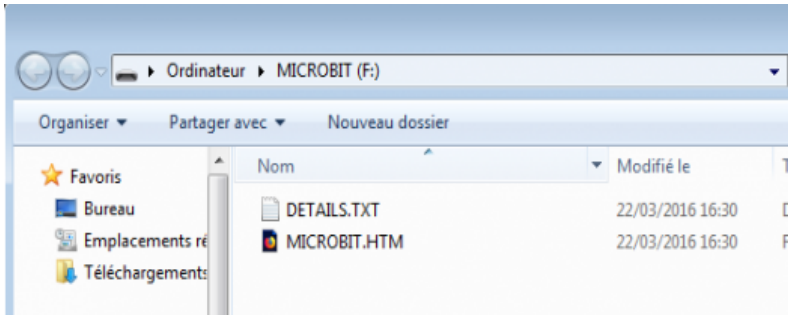
Pour cette raison, on préférera travailler avec un autre éditeur Python à installer sur l'ordinateur (fonctionnant sous Windows, Linux ou Mac) prenant en charge une version de Python dédiée aux microcontrôleurs (appelée MicroPython). Ce logiciel s'appelle Mu-Editor et se trouve sur le site https://codewith.mu.

On choisira sur la page de téléchargement la version adaptée à son système d'exploitation. Nous allons décrire l'installation sous Windows ainsi que celle sous Linux :

3.2.1. *Installation sous Windows :*

- Si on ne connaît pas sa version de Windows (32 ou 64 bits), ouvrir le Panneau de configuration et aller dans Système et Sécurité puis Système. L'information se trouve à la rubrique "Type du système"
- Choisir sur la page de téléchargement la version adaptée : 32 ou 64 bits puis exécuter le fichier pour installer mu-editor. Si plusieurs personnes peuvent utiliser l'ordinateur, penser à cocher l'option qui autorise l'accès du logiciel à tous les utilisateurs. Le détail de

l'installation (également pour les administrateurs réseaux) est donné dans le menu « How to » du site de Mu-Editor
- Il faudra également installer le driver :
  - Télécharger le driver (mbedWinSerial_16466.exe) qui se trouve en principe à cette adresse : http://os.mbed.com/media/downloads/drivers/mbedWinSerial_16466.exe (sinon faire une recherche sur le mot mbedWinSerial, mais ne le télécharger que sur le site os.mbed.com)
  - Connecter la carte BBC Micro:Bit au PC à l'aide du câble USB
  - La carte est vue comme un lecteur contenant deux fichiers



  - Refermer cette fenêtre
  - Installer le driver mbedWinSerial téléchargé

3.2.2. *Installation sous Linux :*

- Sous Linux, on installera mu-editor à l'aide de l'outil de gestion de paquets Python pip :
  - Vérifier dans le gestionnaire de paquets la présence du paquet python3-pip ; en faire éventuellement l'installation. (Ci-dessous un exemple dans Synaptic)

S	Paquet	Version installée	Dernière versic
☐	python3-pil.imagetk-dbg		4.0.0-4
☐	python3-pilkit		1.1.13+dfsg-2
☐	python3-pint		0.7.2-3
☑	python3-pip	9.0.1-2	9.0.1-2
☑	python3-pkg-resources	33.1.1-1	33.1.1-1
☐	python3-pkgconfig		1.2.2-1
☐	python3-pkginfo		1.2.1-1

- On peut alors utiliser l'outil pip. Dans un Terminal, entrer la commande : **pip3 install mu-editor**
- **pip** va alors télécharger tous les fichiers nécessaires au bon fonctionnement de mu-editor et les installer.
- Pas de driver à installer, par contre il faut que l'utilisateur appartienne au groupe dialout pour pouvoir utiliser la connexion USB entre l'ordinateur et la carte :
  - exemple ici pour ajouter l'utilisateur "vincent" (à remplacer par le nom de l'utilisateur !) au groupe dialout :

```
vincent@vincent-VirtualBox:~$ sudo adduser vincent dialout
```

- Attention, l'intégration dans un groupe n'est prise en compte qu'au lancement du compte utilisateur : il faudra donc fermer la session puis la réouvrir
  - On pourra vérifier l'intégration au groupe avec la commande : **groups**

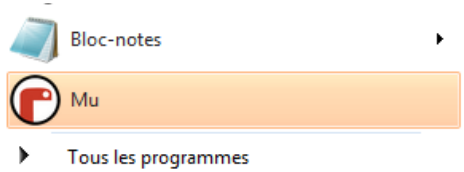
On est prêt maintenant à tester l'association du logiciel mu-editor avec la carte Micro:Bit !

Remarque : sous Debian "Stable" l'installation a échoué pour une raison d'incompatibilité de fichiers. Par contre elle se déroule sans problème dans une version Debian "Unstable" (Debian 10) ou dans une version récente d'Ubuntu (18.04) ou dérivée. Debian 10 sera la version stable de Debian en Mars 2019.

3.3 Premiers pas avec Mu-Editor

3.3.1. *Démarrer mu-editor :*

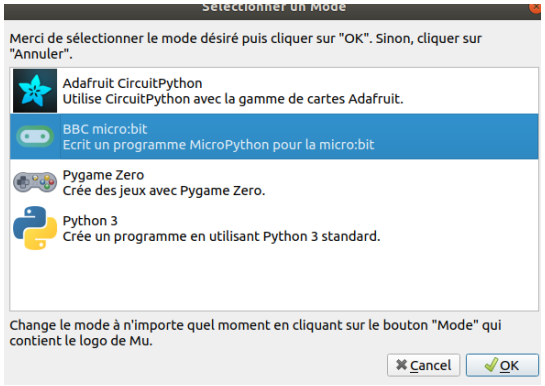
- Sous Windows, on trouvera mu-editor dans la liste des programmes :



- Sous Linux, on entrera dans un Terminal la commande **mu-editor**

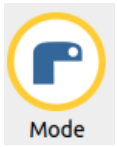
3.3.2. *Découvrir mu-editor :*

Au premier démarrage, une boite de dialogue va s'ouvrir pour demander dans quel mode utiliser mu-editor :

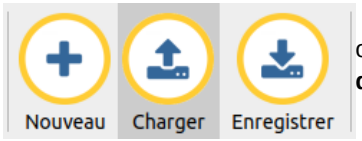


On choisira le mode BBC micro:bit. Comme indiqué en introduction, on pourra à tout moment changer de mode pour repasser dans une console Python classique. Cela se fera à l'aide du bouton Mode.

L'ensemble du menu de mu-editor est réalisé au travers d'icônes séparées en cinq groupes :



Permet de revenir dans la boite de dialogue pour changer le mode d'utilisation de mu-editor... Indispensable si on n'a pas fait le bon réglage à la première mise en route du logiciel !



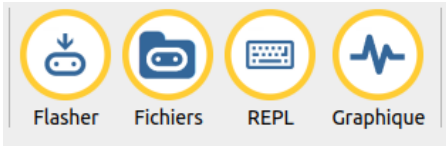
Ce bloc d'icônes concerne les opérations de fichier **sur le disque dur du PC**

- **NOUVEAU** : pour créer un nouveau programme python
- **CHARGER** : pour récupérer un fichier programme enregistré préalablement sur le PC
- **ENREGISTRER** : pour sauvegarder un fichier programme sur le PC. Attention : le logiciel ne propose pas l'option enregistrer sous. Donc si on modifie le contenu d'un fichier déjà nommé, le bouton "Enregistrer" viendra écraser le fichier enregistré sur le disque dur avec le contenu présent à l'écran. La solution consiste alors à cliquer sur le bouton "Nouveau", ce qui ouvre un nouvel onglet nommé "sans titre", et à copier-coller dedans la/les parties de programme que l'on souhaite reprendre pour une nouvelle utilisation

Remarque : lors de l'installation de mu-editor, un dossier "mu-code" a été créé sur le disque dur, directement dans le dossier de l'utilisateur

- Sous Windows, il se trouve en :
  Disque Local (C:)\Utilisateurs\ nom_utilisateur\mu_code\
- Sous Linux : /home/nom_utilisateur/mu_code/

Remarque : pour utiliser l'option **Fichiers** *dans le bloc d'icônes ci-dessous, les programmes devront être stockés à la racine du dossier mu_code et pas dans un sous-répertoire.*



Ce troisième bloc d'icônes concerne les opérations de communication entre le logiciel mu-editor et la carte Micro:Bit :

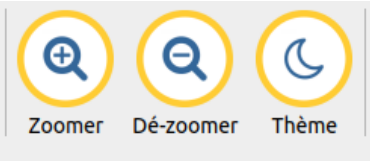
- **FLASHER** : pour programmer dans le microcontrôleur le programme python présent dans la fenêtre d'édition. La LED orange située côté composant de la carte se met à clignoter pendant le chargement du programme dans la mémoire Flash du microcontrôleur. Puis le programme démarre directement sur la carte. (Rque : si on voit sur l'écran de Led un défilement incohérent, c'est qu'il y a une erreur dans le code source...)
- **FICHIERS** : MicroPython dispose sur la carte MicroBit d'un petit système de fichiers (environ 30 kilo octets), permettant de stocker dessus quelques fichiers. On peut réaliser le transfert du PC vers la carte ou inversement. En cliquant sur cette icône "Fichiers" on accède alors à ce système de fichiers.

Pour quitter ce mode il faut recliquer sur l'icône Fichiers

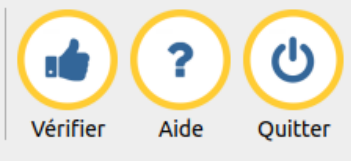
Utilisations possibles de cette option Fichiers :

- stocker un programme Python dans la carte, pour pouvoir le réutiliser sur un autre PC
- Plus intéressant : dans un programme destiné par exemple à de la mesure, il est possible de créer sur la carte, un fichier pour enregistrer des couples date/valeur. On peut alors avec ce système de fichiers transférer le fichier vers le PC pour l'ouvrir dans un tableur par exemple. Cette possibilité sera étudiée plus loin dans ce manuel.

- **REPL** : (Read-Evaluate- Print Loop) : c'est un interpréteur interactif, qui permet d'envoyer vers la carte Micro:Bit des instructions ligne par ligne.
- **GRAPHIQUE** : cette option permet de visualiser graphiquement en temps réel des valeurs mesurées par la carte Micro:Bit



Ce quatrième groupe permet de travailler sur l'aspect du logiciel. L'option ZOOMER sera utile en vidéoprojection en classe pour obtenir à distance une bonne visibilité du code



Dans ce dernier groupe d'icônes, le bouton **VERIFIER** permet de faire une analyse "grammaticale" du code écrit : les fautes seront signalées. Ce vérificateur est plutôt strict, et des "erreurs" pointées ne sont pas toujours rédhibitoires...

3.3.3. Ecriture du code

Après tous ces préparatifs, nous voilà enfin prêts à écrire nos premières lignes de code.

Remarque importante : lorsque l'on écrit un programme en Python, il faut faire très attention à l'indentation du code, indentation que l'on réalisera avec la touche tabulation (ou par 4 espaces consécutifs).

L'indentation se réalise toujours après un double point (« : ») pour délimiter l'entrée dans un bloc du programme. Toutes les lignes de code de ce bloc doivent être alignées sur la même verticale. Ceci est indispensable à l'interpréteur Python, mais nous est aussi très utile pour bien comprendre le code.

Modifier l'indentation d'un bloc modifie complétement la logique du programme !

Ci-dessous l'exemple d'un programme étudié en fin de ce manuel :

```
1  from microbit import *
2
3  display.set_pixel(2, 2, 9)
4  boucle_mesure = False
5
6  while True:
7      while boucle_mesure:
8          print((display.read_light_level(), ))
9          sleep(100)
10         if button_b.was_pressed():
11             boucle_mesure = False
12             display.set_pixel(2,2,9)
13             print('Arret des mesures')
14
15     if button_a.was_pressed():
16         boucle_mesure = True
17         display.clear()
18         print('Mesures :')
19
```

On a représenté par des doubles flèches colorées les indentations à réaliser avec la touche TAB du clavier. Remarquer les lignes pointillées verticales fournies par l'éditeur Mu-Editor pour se repérer facilement dans l'indentation. ***En recopiant les codes d'exemple fournis dans ce manuel, il faudra faire très attention à respecter les indentations.***

4 Découvrir la carte BBC Micro:Bit

Dans ce chapitre nous allons mettre en œuvre des petits programmes pour :

- utiliser quelques unes des ressources disponibles sur la carte Micrro:Bit
- (re)voir quelques éléments du langage Python

4.1 Importation du module de fonctions microbit

Un programme écrit en Python s'appuie généralement sur des fonctions standards du langage (appelées *fonctions natives* ou *built-in functions*)mais aussi sur d'autres fonctions utiles dans des cas spécifiques et regroupées dans des **modules**. Il existe par exemple le module math qui permet d'utiliser dans un programme de nombreuses fonctions mathématiques (trigonométriques, logarithmiques, exponentielles ...) ou des constantes (pi, e...).

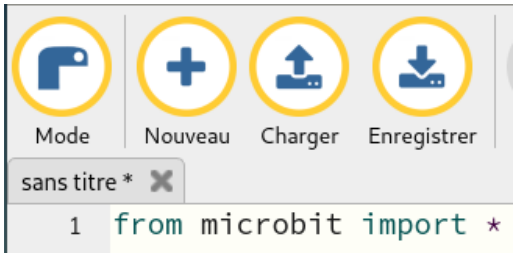
Ces modules doivent être importés en début de programme pour que les fonctions qu'ils contiennent soient utilisables dans le reste du corps du programme. Cela se réalise par une simple ligne comme par exemple :

from math import *

(* signifie que l'on importe toutes les fonctions du module math)

Un module spécifique au fonctionnement de la carte Micro:Bit est nécessaire : il s'agit du module "microbit".

Ainsi, pour pouvoir utiliser la carte BBC Micro:Bit, tous nos programmes commenceront par l'importation de ce module :



Toutes les fonctions fournies par ce module sont décrites à l'adresse suivante :

https://microbit-micropython.readthedocs.io/en/latest/index.html

Nous en utiliserons quelques unes dans les exemples proposés. Pour exploiter davantage la carte Micro:Bit, il pourra être nécessaire de s'y référer.

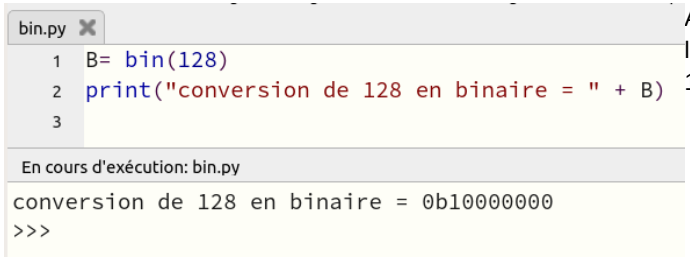
4.2 Un peu de vocabulaire

Fonction, procédure, méthode, paramètres, arguments … quelques précisions :

Fonction : Comme évoqué au paragraphe précédent Python possède un certain nombre de fonctions natives. Prenons la fonction bin(x) qui transforme un nombre entier en une chaîne de caractère représentative de la conversion en binaire de ce nombre. Mu-Editor a été utilisé en mode Python 3 :

Dans la définition de la fonction bin, x représente le *paramètre* (unique) que prend la fonction bin.

Complétons, puis exécutons le programme :



A la ligne 1, on réalise *l'appel de la fonction* bin avec *l'argument* 128.

Cette fonction retourne une « valeur » (ici une chaîne de caractère) que l'on peut affecter à une variable (ici B)

Ainsi une fonction pourra être utilisée dans une instruction du type :

variable = fonction (argument1,argument2,…)

On retrouve cette même définition d'une « vraie » fonction dans d'autres langages.

Par contre, on rencontre en Python des fonctions qui ne retournent aucune valeur, que l'on nomme néanmoins fonction, alors que dans d'autres langages elles auraient été appelées *procédures*.

Exemple : dans les programmes qui tournent sur microcontrôleur, on verra que l'on doit assez souvent introduire des délais de temporisation. On utilisera alors une fonction nommée sleep(time) (le paramètre time sera exprimé en milliseconde). On écrira alors une ligne d'instruction du type :

sleep(500)

pour mettre en pause le programme pendant une demi seconde.

Enfin, Python permet de faire de la programmation orientée objet (POO). Pas de crainte à avoir : nous ne ferons qu'utiliser des objets préexistants ce qui n'a rien de bien compliqué comme on va le voir dans le prochain paragraphe avec l'utilisation de l'afficheur 5x5 Leds.

Les fonctions applicables à un objet portent le nom de **méthode**.

4.3 Gestion de l'afficheur 5x5 Leds

4.3.1. Allumage d'une Led

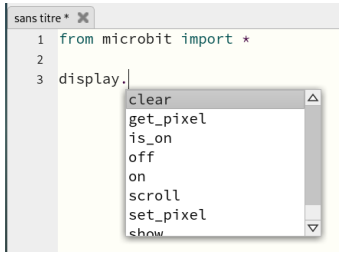
L'afficheur 5x5 Leds est perçu par nous comme étant un objet réel.

Le module microbit fournit un objet au sens Python, nommé "**display**" sur lequel on peut appliquer des méthodes en suivant la syntaxe :

objet.méthode

dans le but d'agir sur l'objet réel qu'est l'afficheur

Mu-editor apporte une aide bien pratique avec l'auto-complétion du code : quand on tape **display suivi d'un point**, mu-editor nous montre toutes les méthodes que l'on peut appliquer à l'objet display :



Il suffit de sélectionner la méthode désirée par un double clic. Puis, l'ouverture d'une parenthèse fait apparaître une *aide contextuelle* qui vient rappeler les différents paramètres que l'on doit entrer :

set_pixel(x,y,b) permet de régler la luminosité b d'une Led de coordonnées x et y sur une échelle de valeurs allant de 0 à 9.

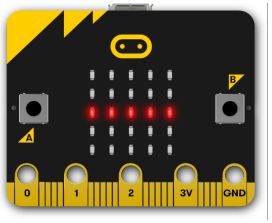
A titre d'exemple, entrer le code ci-dessous (Mu-Editor devra être dans le Mode BBC Micro:Bit) :

```
from microbit import *

display.set_pixel(0,2,9)
```

Enregistrer le fichier en lui donnant un nom explicite ... puis cliquer sur le bouton **FLASHER** : la mémoire Flash du microcontrôleur reçoit ce code puis le programme démarre : on doit voir la Led centrale briller fortement.

Exercice 4.3.1_1:

Histoire de jouer avec les coordonnées des Leds, écrire le code nécessaire pour obtenir le signe moins (-) qui devra occuper l'intégralité de l'afficheur :



Remarque : pour comprendre le système de coordonnées (x,y) de l'afficheur, on pourra s'entraîner plus rapidement en utilisant l'interpréteur interactif : cliquer sur le bouton REPL, puis dans l'interpréteur situé en bas de fenêtre, entrer des lignes d'instruction comme celle ci-dessous (validées par la touche Entrée du clavier) :

display1.py

```
1  from microbit import *
2
3  display.set_pixel(2,2,9)
```

BBC micro:bit REPL

```
MicroPython v1.9.2-34-gd64154c73 on 2017-09-01;
Type "help()" for more information.
>>>
MicroPython v1.9.2-34-gd64154c73 on 2017-09-01;
Type "help()" for more information.
>>>
>>> display.set_pixel(1,0,5)
>>>
```

(on verra ici la deuxième Led de la première ligne s'éclairer moyennement)

Rappel : on sort du mode REPL en recliquant sur ce même bouton

Une façon simple d'avoir répondu au premier problème posé est d'utiliser une suite d'instructions display.set_pixel() comme celle ci-dessous :

```
from microbit import *

display.set_pixel(0,2,9)
display.set_pixel(1,2,9)
display.set_pixel(2,2,9)
display.set_pixel(3,2,9)
display.set_pixel(4,2,9)
```

C'est une façon un peu "laborieuse" et pas très "élégante" du point de vue du code.

A y regarder de plus près on constate de ligne en ligne que **x croit de 0 à 4 pendant que y reste constant à 2**

Ces 5 lignes pourraient être remplacées par une seule :

display.set_pixel(i , 2 ,9)

à condition que i puisse prendre successivement les valeurs 0, 1, 2, 3 et 4

C'est possible en réalisant dans le programme une **boucle**

4.3.2. La boucle While :

```
while condition:
        instruction 1 à réaliser dans la boucle
        instruction 2 à réaliser dans la boucle
        etc...
1 ère instruction à réaliser en sortant de la boucle
```

... qui signifie *"tout le temps que la **condition** est vraie, exécute le code dans le bloc indenté ci-dessous"*

Appliquons la à notre programme qui devient alors :

```
from microbit import *
i = 0
while i < 5 :
    display.set_pixel(i, 2, 9)    # x va varier de 0 à 4 ; y fixé à 2
    i = i + 1
```

Ex_432.py

Explication :

On commence par déclarer une **variable** i à laquelle **on affecte** la valeur 0.

(le signe = n'est pas un signe d'égalité mais d'affectation, qui signifie « on affecte la valeur qui est à droite du symbole = à la variable écrite à gauche»)

On teste si i est strictement inférieur à 5 : c'est vrai, donc on rentre dans le bloc d'instructions avec i valant 0

La ligne suivante va donc allumer la Led de coordonnées (0,2) avec la luminosité maximum (9)

La ligne d'instruction suivante i = i + 1 est déroutante lorsque l'on démarre la programmation. Il faut la comprendre de la façon suivante :

on affecte la valeur i + 1, c'est à dire 0 + 1, donc 1

à la variable située à gauche du signe = , donc à la variable i

Lorsque la ligne d'instruction i = i + 1 aura été exécutée, i ne vaudra plus 0 mais 1 : on a réalisé une *incrémentation* de la variable i

C'était la dernière ligne d'instruction dans ce bloc, on remonte donc au test while i <5 qui est de nouveau vrai puisque i vaut 1.

Cela déclenche l'allumage d'une nouvelle Led de coordonnées (1,2), puis nouvelle incrémentation de la variable qui prend alors la valeur 2, etc. jusqu'à ce que i prenne la valeur 4. La boucle est parcourue une dernière fois : à la suite de l'incrément i prend la valeur 5. La condition i < 5 n'étant plus vérifiée la boucle n'est plus parcourue.

Rque : si on a déjà fait de la programmation, on se demande sans doute pourquoi ne pas utiliser plutôt une boucle « à compteur » de type « for ». Exemple en C :


```
for (i = 0 ; i ≤ 4 ; i++)
        {


        }
```

qui nécessite moins de code(une seule ligne contre 3 avec la boucle while : initialisation de la variable, test d'entrée de boucle et incrémentation dans la boucle…

Deux raisons à ce choix de présentation :

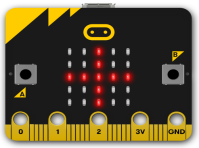
1. ***on va avoir besoin rapidement de réaliser des boucles infinies, et c'est mieux de découvrir la boucle while sur un exemple de boucle finie***
2. ***L'instruction « for » existe aussi en Python (sous la forme « for...in...) », mais elle a un rôle plus puissant (elle permet de travailler sur des séquences comme par exemple des listes). On la mettra en œuvre plus tard. Une de ses applications est de pouvoir aussi faire une boucle à compteur :***

```
for i in range(5):
```

la fonction range(5) commence par créer une liste de 5 entiers de 0 à 4 , et cette liste est parcourue avec l'instruction for, donnant successivement à i la valeur des éléments de la liste

Exercice 4.3.2 1:

Écrire un second programme pour obtenir sur l'écran le signe plus(+) qui devra occuper l'intégralité de l'afficheur.



Ce programme devra utiliser lui aussi une boucle While

Exercice 4.3.2 2 :

On souhaite dans un même programme :

- **afficher le signe + pendant 1 seconde**
- **effacer l'écran et le voir rester ainsi pendant pendant 0,2 seconde**

- **et enfin afficher le signe -**

Écrire le code nécessaire respectant le cahier des charges défini ci-dessus. Pour cela, on pourra bien sûr réutiliser du code déjà écrit mais on aura aussi besoin de savoir exploiter les informations suivantes :

- une temporisation (ou délai) se réalise dans un programme en MicroPython avec la fonction :

sleep(n)

dans laquelle n représente le nombre de milliseconde de délai

- l'effaçage de l'écran se fait avec la méthode :

display.clear()

Réaliser une boucle infinie :

On veut améliorer le programme précédent pour afficher en permanence la succession des signes + et – avec le même cahier des charges que précédemment, à savoir : l'affichage des signes pendant une seconde, l'effacement de l'écran pendant 0,2 s

Pour cela on va de nouveau utiliser une boucle While dans laquelle va être incluse le programme précédent.

Ici il nous faut donc une condition qui soit toujours vraie pour que la boucle tourne indéfiniment.

Rque : un microcontrôleur n'est en principe pas programmé pour réaliser une suite d'actions puis s'arrêter. On inscrit donc toujours le programme écrit pour un microcontrôleur dans une boucle infinie :

Une possibilité serait de déclarer une variable booléenne comme étant Vraie (True), variable inutilisée dans le reste du programme. elle restera donc toujours Vraie. A chaque entrée dans la boucle on teste si elle est toujours vraie. Comme elle le sera le programme tournera indéfiniment dans la boucle :

```
tournicoti = True
while tournicoti == True :
        instruction 1 à réaliser dans la boucle
        instruction 2 à réaliser dans la boucle
        etc...
```


Remarquer le double signe == dans le test de condition. On retrouvera ce symbole dans un autre test conditionnel (if) un peu plus loin dans ce manuel.

Python permet une écriture plus concise qui peut être néanmoins déroutante. L'idée est simple : à quoi bon déclarer et tester une variable qui ne change pas. Le plus court est alors d'écrire :

```
while True :          #qui signifie tout le temps que Vrai est Vrai … !!
        instruction 1 à réaliser dans la boucle
        instruction 2 à réaliser dans la boucle
        etc...
```

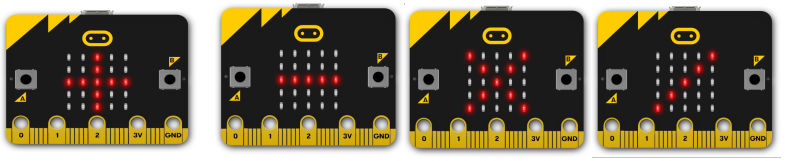
Ci-dessous le code :

```
from microbit import *

while True:
    i = 0
    # affichage du signe + :
    while i < 5:
        display.set_pixel(i,2,9)
        display.set_pixel(2,i,9)
        i = i + 1
    # réalisation du délai et de l'effaçage :
    sleep(1000) #1000ms = 1 seconde !
    display.clear()
    sleep(200)
    # affichage du signe - :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        i = i + 1
    # réalisation du délai et de l'effaçage :
    sleep(1000) #1000ms = 1 seconde !
    display.clear()
    sleep(200)
```

La séquence d'instructions qui n'était exécutée qu'une seule fois est maintenant incluse dans une boucle while infinie. Remarquer de

nouveau l'indentation du code réalisé à l'aide de la touche tabulation (on a ici entouré les différentes blocs de code des boucles while).

Exercice 4.3.2_3 : Compléter le programme ci-dessus pour pouvoir afficher au même rythme, la succession des symboles +, - , x et / :



4.3.3. Écriture d'une fonction :

Une fonction sans paramètres :

Dans le code précédent on s'aperçoit qu'à la fin de l'affichage de chaque signe, on est amené à réécrire les trois mêmes lignes de code pour réaliser la temporisation demandée.

Il est alors judicieux de créer une fonction, appelons la delai() constituée de ces trois lignes de code et que l'on appellera à la suite de l'affichage de chaque signe. Cela donne :

```
from microbit import *

def delai():
    # réalisation du délai et de l'effaçage :
    sleep(1000)    #1000ms = 1 seconde !
    display.clear()
    sleep(200)                #200ms = 0,2 seconde !

while True:

    # affichage du signe + :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        display.set_pixel(2,i,9)
        i = i + 1
```



```
delai()          # appel de la fonction delai()

# affichage du signe - :
i = 0
while i < 5:
    display.set_pixel(i,2,9)
    i = i + 1

delai()          # appel de la fonction delai()

# affichage du signe x :
i = 0
while i < 5:
    display.set_pixel(i,i,9)
    display.set_pixel(4-i,i,9)
    i = i + 1

delai()          # appel de la fonction delai()

# affichage du signe / :
i = 0
while i < 5:
    display.set_pixel(4-i,i,9)
    i = i + 1

delai()          # appel de la fonction delai()
```

Pour définir une fonction, on utilise le mot réservé « def » suivi du nom que l'on veut donner à la fonction, suivi des deux points annonçant un bloc qui devra être indenté et contenir les lignes d'instructions de la fonction

Cette « fonction » délai() ne prend aucun paramètre, ne renvoie aucun résultat. C'est juste une suite d'instructions à réaliser. Dans d'autres langages elle serait nommée « procédure ». C'est en tout cas un moyen d'alléger et de rendre plus lisible le programme.

Autre intérêt : si on trouve que la temporisation de 1 seconde est insuffisante, il suffit d'aller dans la définition de la fonction delai() et de changer la ligne concernée :

```
def delai():
    # réalisation du délai et de l'effaçage :
    sleep(2000)    # 2000ms = 2 secondes !
    display.clear()
    sleep(200)                # 200ms = 0,2 seconde !
```

Au lieu de devoir changer en quatre endroits cette valeur, il a suffit d'une seule fois !

Une fonction avec paramètre(s) :

… tout cela est très bien, mais si on veut une temporisation spécifique pour chacun des signes +, -, x et / ?

C'est là qu'arrive une possibilité très intéressante : définir une fonction avec un ou plusieurs paramètres.

Reprenons la définition de la fonction et ajoutons un paramètre que l'on va appeler dt, qui permettra de choisir individuellement la durée de temporisation :

```
def delai(dt):
    # réalisation du délai et de l'effaçage :
    sleep(dt)      # dt = délai en millisecondes
    display.clear()
    sleep(200)
```

Désormais, l'appel de la fonction se fera à chaque fois en précisant la durée de temporisation :

```
while True:

    # affichage du signe + :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        display.set_pixel(2,i,9)
```



```
        i = i + 1

    delai(500)
```

Exercice 4.3.3_1 : compléter ce code pour avoir une temporisation :

- **d'une seconde pour le signe -**
- **de deux secondes pour le signe x**
- **et de quatre secondes pour le signe /**

Exercice 4.3.3.2 :

On souhaite maintenant pouvoir choisir non seulement la temporisation d'affichage, mais aussi celle de l'écran effacé. Faire les modifications nécessaires dans le programme.

Rque : comme toujours, lorsque l'on veut tester une modification dans un programme, faire des essais avec différentes valeurs !

4.3.4. Créer un module de fonctions

La fonction créée dans le paragraphe précédent n'a guère d'autre utilité que dans le programme dans lequel elle a été conçue. Il peut arriver que des fonctions créées dans un programme puissent de nouveau être intéressantes à utiliser dans d'autres programmes. Il est alors être judicieux de placer ces fonctions dans un fichier externe au programme. Ce fichier constituera un module de fonctions pers
On connaît déjà un tel module : le module microbit dont on importe l'intégralité à chaque début de nos programmes avec la ligne d'instruction :

```
from microbit import *
```

De ce module, on connaît par exemple pour l'afficheur 5x5 Leds les méthodes suivantes :

set_pixel(x,y,n) pour allumer une Led
clear() pour éteindre toutes les Leds

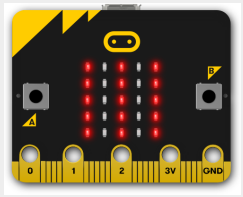
Il en existe d'autres pour afficher une imagette, faire défiler un message, quelques autres pour gérer l'état de l'afficheur.
On se propose ici de créer de nouvelles fonctions graphiques pour l'afficheur 5x5 Leds et de les inclure ensuite dans un module de fonctions. Ce seront

des « primitives graphiques » c'est à dire des fonctions basiques de dessin qui permettront de composer plus facilement des dessins plus complexes.

Commençons par créer une fonction qui allume l'intégralité d'une colonne de Leds d'abscisse x, et testons la dans un même programme :

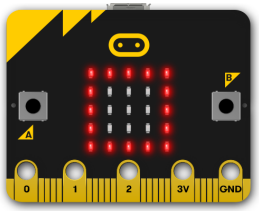
```
from microbit import *

def colonne(x):
    # allumage d'une colonne de Leds
    # attention : x compris entre 0 et 4
    i = 0
    while i < 5:
        display.set_pixel(x,i,9)
        i = i + 1

while True:
    colonne(0)
    colonne(2)
    colonne(4)
```

On doit obtenir ceci :



Exercice 4.3.4_1 : créer une fonction que l'on appellera ligne(y) pour allumer une ligne (horizontale) de 5 Leds d'ordonnée y et la tester dans un même programme.

Exercice 4.3.4_2 : écrire un programme utilisant les fonctions ligne(y) et colonne(x) pour obtenir un grand carré :



… tout cela est bien joli, mais on ne va pas pouvoir faire grand-chose au niveau dessin avec ces fonctions : ce serait mieux de pouvoir allumer des petits segments verticaux ou horizontaux de longueurs quelconques.

Rque : il est souvent utile de faire comme ici un premier travail préparatoire qui permet de valider le principe, et de voir exactement ce que l'on peut en

faire et quelles en sont les limites. A la suite de quoi on peut décider d'affiner le projet…ce qui va être fait avec les exercices qui suivent.

Exercice 4.3.4_3 : Redéfinir la fonction colonne de façon à ce qu'elle prenne 3 paramètres :

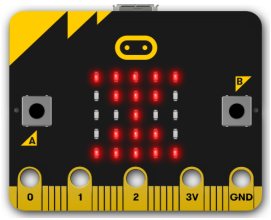
- **les coordonnées x1 et y1 du point de départ haut de la colonne**
- **l'ordonnée y2 de l'autre extrémité**

colonne(x1,y1,y2)

Exercice 4.3.4_4 : … et bien sûr maintenant la même chose pour tracer des segments horizontaux avec une fonction ligne redéfinie par :

ligne(x1,y1,x2)

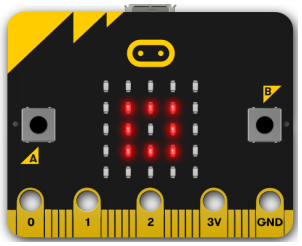
- **les coordonnées x1 et y1 du point de départ gauche de la ligne**
- **l'abscisse x2 de l'autre extrémité**

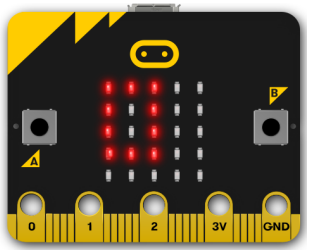


Utiliser cette fonction pour réaliser cette image de sablier.

Exercice 4.3.4_5: les segments que l'on sait tracer peuvent maintenant nous servir à afficher des carrés ou des rectangles.

1. **Écrire les programmes permettant de réaliser l'affichage des figures ci-dessous :**





2. **Pour simplifier l'affichage d'une telle forme (carré ou rectangle), on souhaite réaliser la fonction suivante :**

rectangle(x1,y1,x2,y2)

dans laquelle

- **x 1et y1 sont les coordonnées du sommet haut-gauche**
- **x2 et y2 les coordonnées du sommet opposé**

Écrire le code nécessaire à la définition de cette fonction et le tester pour obtenir à nouveau le carré ou le rectangle présentés ci-dessus.

Nous sommes maintenant en possession de nouvelles fonctions graphiques. Pour pouvoir les réutiliser facilement dans d'autres programmes, on va créer un module de fonction personnalisé. C'est un fichier de type script Python, qui doit donc avoir l'extension « .py ». Il faut lui donner un nom compréhensible. Appelons-le « affleds.py »

voici son contenu :

```
from microbit import *

def ligne(x1,y1,x2):
    # allumage d'une ligne de Leds
    # x1 et y1 : point de départ gauche
    # x2 point d'arrivée
    i = x1
    while i <= x2:
        display.set_pixel(i,y1,9)
        i = i + 1

def colonne(x1,y1,y2):
    # allumage d'une colonne de Leds
    # x1 et y1 : point de départ haut
    # y2 point d'arrivée
    i = y1
    while i <= y2:
        display.set_pixel(x1,i,9)
        i = i + 1

def rectangle(x1,y1,x2,y2):
    # x1, y1 coordonnées du sommet supérieur gauche
    # x2, y2 coordonnées du sommet inférieur droit
```



```
    ligne(x1,y1,x2)
    colonne(x1,y1,y2)
    ligne(x1,y2,x2)
    colonne(x2,y1,y2)
```

Rque : il est fortement conseillé de bien documenter un module de fonctions pour pouvoir le réutiliser facilement plusieurs semaines après sa conception…

Enregistrer ce fichier dans le dossier « mu-editor » utilisé par défaut pour le stockage de nos programmes.

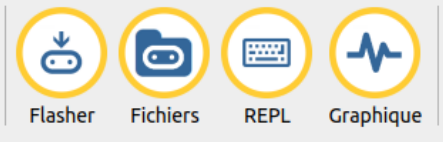
Rque : lors de l'importation d'un module, Python recherche le fichier tout d'abord dans le répertoire courant.

On se propose d'utiliser ce module dans le petit programme suivant qui va nous permettre une petite animation graphique autour de deux carrés. Appelons le « test_affleds.py » :

```
from affleds import *

while True:
    rectangle(0,0,4,4)
    sleep(150)
    display.clear()
    rectangle(1,1,3,3)
    sleep(150)
    display.clear()
```

Tout semble correct…cependant, à l'exécution cela ne fonctionne pas !
En fait, il faut se souvenir que le programme que l'on vient d'écrire tourne, non pas sur l'ordinateur de travail, mais sur le microcontrôleur de la carte BBC micro:bit ! C'est donc en cet endroit que doit se trouver notre module de fonctions affleds.py !!!

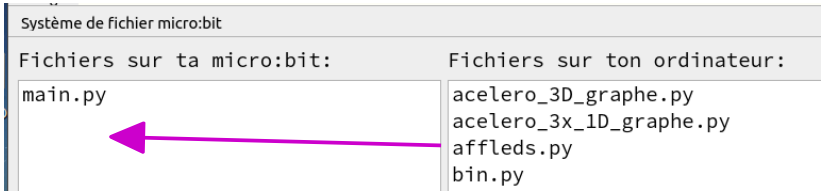
On doit donc accéder au système de fichiers de la carte : cela se fait avec le bouton Fichiers :



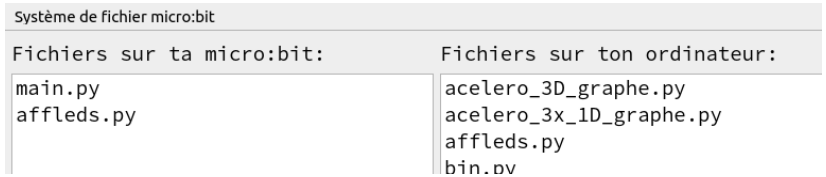
Un panneau s'ouvre montrant deux zones : à gauche les fichiers présents sur la carte micro:bit, à droite ceux dans le répertoire mu-editor de l'ordinateur.

Le système de fichiers de la carte micro:bit ne contient qu'un seul programme nommé « main.py » (explication un peu plus loin…)

Dans la partie de droite, sélectionner le module de fichiers « affleds.py » et le déplacer dans le système de fichiers de la carte :



La led orange de la carte se met à clignoter, preuve qu'un fichier est en train d'être chargé dans sa mémoire. Après quelques instants on obtient :



Rque : le module de fichiers est toujours présent sur le PC : il a seulement été copié dans la mémoire de la carte.

Recliquer sur l'icône Fichiers pour sortir de ce mode et flasher à nouveau le code de notre programme de test. Le programme fonctionne bien maintenant !

Retournons dans le système de fichiers en cliquant sur l'icône Fichiers : on voit toujours sur la carte le fichier « main.py » et notre module de fichiers. Mais au fait, et notre programme de test « test_affleds.py » : où est-il ???

Essayons tout d'abord de voir ce qu'est ce fichier « main.py » qui se trouve sur la carte.

Déplacer à la souris ce fichier main.py vers le système de fichiers du PC. Après quelques instants il sera disponible.

Quitter le mode Fichiers et cliquer sur le bouton Charger : sélectionner le fichier main.py (que l'on doit trouver maintenant dans le dossier mu-editor), ce qui va l'ouvrir dans notre éditeur :

Le fichier « main.py » est en fait la copie exacte du dernier fichier programme que l'on a flashé dans la mémoire de la carte (donc ici une copie de test_affleds.py).

Explication : la carte micro:bit dispose d'un chargeur de programme qui va systématiquement chercher au démarrage un programme du nom de main.py (ce qui signifie « programme principal »). Lorsque l'on charge un programme avec le bouton flasher, c'est le code contenu dans ce fichier programme qui sert de contenu au fichier main.py.

S'il y a des dépendances (un module de fonction personnalisé, ce fichier doit donc être au même niveau d'arborescence que main.py … d'où l'obligation que l'on eue de placer affleds.py dans le système de fichiers de la carte micro:bit). Les modules standards ou le module microbit sont eux, déjà présents avec l'interpréteur Python.

Exercice 4.3.4_5 : Exercice de synthèse

Le troisième paramètre de la méthode display.set_pixel(x,y,b) permet de régler la luminosité du pixel (x,y).

Les trois fonctions graphiques que l'on a créées dans le module affleds.py on virtuellement perdu cette possibilité de réglage puisqu'à chaque fois, on a mis la luminosité à la valeur maximum (9).

Réécrire les fonctions de ce module pour avoir accès au réglage de la luminosité (que l'on notera lum) d'un segment vertical, horizontal ou d'un rectangle, avec une valeur par défaut égale à 9 mais restant réglable par ailleurs (relire la remarque donnée dans le corrigé de l'exercice 4.3.3.2)

Elles auront donc a forme suivante :

ligne(x1,y1,x2,lum=9)
colonne(x1,y1,y2,lum=9)
rectangle(x1,y1,x2,y2,lum=9)

Exemple d'utilisation : lire le programme suivant et comprendre ce qu'il va réaliser avant de le flasher dans la mémoire de la carte :

```
from affleds import *

while True:
    i = 0
    while i<9:
        rectangle(0,0,4,4,i)
        rectangle(1,1,3,3,i)
        display.set_pixel(2,2,i)
        i = i + 1
        sleep(100)
```



```
    i = 8
    while i>0:
        rectangle(0,0,4,4,i)
        rectangle(1,1,3,3,i)
        display.set_pixel(2,2,i)
        i = i - 1
        sleep(100)
```

4.3.5. Les autres méthodes de l'objet display

Nous avons vu pour l'instant deux méthodes applicables à l'objet display :

display.set_pixel(x,y,n) et display.clear()

display.get_pixel(x,y) permet de savoir quelle est la luminosité (entre 0 et 9) de la Led située en (x,y)

exemple : Cliquer sur REPL et taper dans l'interpréteur interactif : display.get_pixel(2,2) renvoie la valeur 9 si la Led centrale est allumée au maximum.

display.scroll permet de faire défiler un message tandis que

display.show permet d'afficher un message, une image ou une séquence d'images. Des images sont déjà prêtes à être utilisées :

https://microbit-micropython.readthedocs.io/en/latest/tutorials/images.html

Nous utiliserons ces méthodes un peu plus loin dans ce manuel. En attendant, le lecteur curieux pourra vouloir les tester simplement :

Aller dans l'interpréteur interactif (REPL) pour voir la différence de comportement :

display.scroll('ABCDE')

puis

display.show('ABCDE')

Des paramètres complémentaires peuvent être fournis lors de l'utilisation de ces fonctions :

- ***delay= une valeur en milliseconde : pour modifier la vitesse de l'animation***
- ***wait= True ou False : pour obliger ou pas le programme à attendre la fin de l'animation avant de poursuivre l'exécution des lignes de code qui suivent***
- ***loop= True ou False : pour faire recommencer l'animation ou pas une fois arrivée à son terme***
- ***clear= True ou False pour effacer l'écran à la fin de l'animation (uniquement pour display.show)***

La méthode :

display.read_light_level()

permet d'utiliser l'afficheur comme capteur de lumière et sera utilisée plus loin dans ce manuel.

Trois autres méthodes sont encore disponibles :

display.on()
display.off()
display.is_on()

La première pour activer l'afficheur, la seconde pour le désactiver et la troisième pour redemander s'il est activé ou non.

Utilité : le connecteur d'extension de la carte BBC micro:bit partage quelques unes de ses broches avec l'afficheur. Pour éviter un conflit, il est nécessaire de désactiver l'afficheur (cela peut être que momentanément au cours de l'exécution du programme) et de le réactiver si nécessaire. Ces méthodes sont utilisées dans des projets de plus grande envergure.

Pour plus de détails aller sur la page :

https://microbit-micropython.readthedocs.io/en/latest/display.html

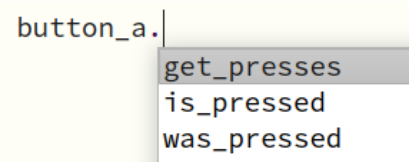
4.4 Gestion des boutons poussoirs

4.4.1. Méthodes associées

Deux boutons poussoirs sont disponibles pour l'utilisateur :

- le bouton A à gauche de l'afficheur est géré par l'objet **button_a**
- le bouton B à droite de l'afficheur est géré par l'objet **.... button_b**

les méthodes disponibles :



- **button_a.is_pressed()** : à utiliser dans un test, cette méthode renvoie la valeur booléenne True si le bouton est pressé et False dans le cas contraire
- **button_a.was_pressed()** : à utiliser dans un test, cette méthode permet de savoir si le bouton a été pressé depuis le démarrage ou depuis le dernier appel de cette méthode. L'appel à cette méthode remet à zéro l'état logiciel du bouton : il faudra donc que le bouton soit de nouveau pressé pour que cette méthode retourne à nouveau la valeur True
- **button_a.get_presses()** : à utiliser dans une ligne de code telle que :

N = button_a.get_presses()

N représente alors le nombre de fois que le bouton a été pressé. L'utilisation de cette méthode remet à zéro le compteur.

Mettons en œuvre la méthode qui sera certainement la plus utilisée, celle qui consiste à savoir si un bouton est pressé.

4.4.2. L'instruction conditionnelle : IF

Pour savoir si un bouton a été pressé, on va utiliser une instruction conditionnelle : IF (= "Si")

Regardons ce que donnerait une version francisée des possibilités offertes :

- Pour un test unique (aucune autre alternative n'est traitée) :

```
Si (condition vraie):
    faire ceci
```

- Pour un test unique (toutes les autres alternatives sont traitées) :

```
Si (condition vraie):
    faire ceci
Sinon:
    faire celà
```

- Pour un test multiple (ici on teste trois conditions, avec en plus le traitement de toutes les autres alternatives) :

```
Si (condition1 vraie):
    faire ceci
SinonSi (condition2 vraie):
    faire cela
SinonSi (condition3 vraie):
    faire ça
Sinon:
    faire cette chose
```

Pour rédiger le code en Python il suffira de remplacer :

- **Si par if**
- **Sinon par else**
- **SinonSi par elif (contraction de else if)**

Remarques :

- ***comme toujours en Python, on fera attention à l'indentation des blocs de code (à réaliser avec une tabulation)***
- ***On peut imbriquer plusieurs tests de deux façons pour vérifier que deux conditions sont satisfaites pour déclencher une action :***
  - ***en imbriquant deux tests***

```
if (condition1):
    if (condition2):
        Faire ceci
```

  - ***en liant les deux conditions dans le même if à l'aide de l'opérateur logique "and" (= "et")***

```
if (condition1) and (condition2):
    Faire ceci
```

- ***On peut réaliser une même action lorsque l'une ou l'autre de deux conditions est réalisée à l'aide l'opérateur logique "or" (= "ou") :***

```
if (condition1) or (condition2):
    Faire ceci
```

Exemple : On a évoqué l'existence de la méthode display.show pour l'afficheur. Cette méthode permet d'afficher des caractères ou des images.

Ci-dessous le code qui affiche la lettre A si on appuie sur le bouton A et la lettre B si on appuie sur le bouton B.

L'affichage d'une lettre (A par exemple) se fait simplement de la façon suivante :

display.show('A')

Pour savoir si le bouton A est appuyé, on utilise la méthode :

button_a.is_pressed()

qui renvoie la valeur booléenne

- True si le bouton A est appuyé
- False dans le cas contraire

L'utilisation de l'instruction conditionnelle if, associée à cette méthode permet alors d'orienter le programme selon le bouton appuyé :

```
from microbit import *

display.show('?')

while True:  # création d'une boucle infinie
    if button_a.is_pressed():
        display.show('A')
    elif button_b.is_pressed():
        display.show('B')
```

Explication : dans la première ligne d'instruction de la boucle infinie, la condition évaluée est **button_a.is_pressed()** :

- Si le résultat est **True** (le bouton A est appuyé au moment du test)) alors on rentre dans le bloc indenté qui affiche sur l'écran la lettre A.
- Si le résultat est **False** (le bouton A n'était pas appuyé au moment du test) alors on passe à la ligne d'instruction de même indentation c'est à dire la ligne **elif button_b.is_pressed()** qui traite de la même façon le bouton B.

Quel que soit le résultat de chacun de ces tests, on reviendra toujours à la ligne while True qui permet cette boucle infinie.

On est typiquement ici dans de la programmation évènementielle : le programme ne fait strictement rien d'autre qu'attendre un évènement : l'appui sur un des boutons du système, évènement auquel il réagit.

Rque : si la boucle infinie effectue par ailleurs un traitement long avant de revenir au test des deux boutons alors l'appui sur l'un des boutons peut avoir été « raté », non pas par l'utilisateur, mais par le programme. A titre d'illustration, introduisons une pause de plusieurs secondes dans le programme précédent :

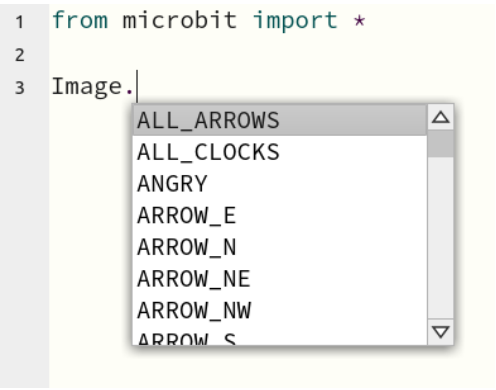
```
from microbit import *

display.show('?')

while True:  # création d'une boucle infinie
    if button_a.is_pressed():
        display.show('A')
    elif button_b.is_pressed():
        display.show('B')
    sleep(10000)    # fais dodo pendant 10 secondes !
```

Très peu de chance de voir un appui bref sur l'un des boutons être pris en compte : il faut presser l'un des boutons pendant plusieurs secondes (au maximum une dizaine de secondes si on est malchanceux...) pour que l'évènement soit pris en compte. Dans une telle situation, il faudra plutôt se tourner vers la méthode button_a.was_pressed()

On a évoqué la possibilité d'utiliser la méthode display.show() pour afficher des images. MicroPython fourni des images prêtes à être utilisées avec l'afficheur 5x5 Leds.

On les obtient avec : Image.nom_de_l_image. On peut en obtenir la liste comme indiqué ci-contre ou à l'adresse

https://microbit-micropython.readthedocs.io/en/latest/tutorials/images.html

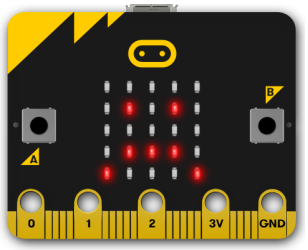


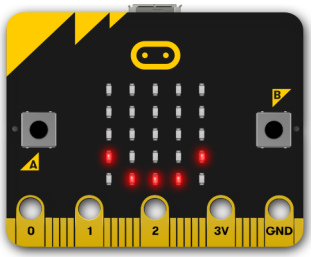
Pour afficher une image (par exemple une flèche orientée vers l'Est) :

```
display.show(Image.ARROW_E)
```

Exercice 4.4.2_1 : Écrire un programme qui affiche une image au démarrage, image qui sera changée ensuite selon le bouton appuyé.

Par exemple avec les images SAD et SMILE :





Exercice 4.4.2_2 : Lire le programme suivant et dire ce qui se passera lors de son exécution :

- **au démarrage**
- **lors de l'appui sur le bouton A ou sur le bouton B**

```
from microbit import *

i = 5

while True:
    display.show(i)
    if i >0 and button_a.is_pressed():
        i = i - 1
    if i<9 and button_b.is_pressed():
        i = i + 1
```

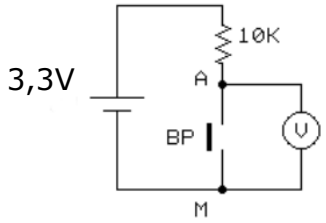
Taper et lancer ce programme. Qu'observe-t-on ?

Rque : il y a une explication à ce problème. Un peu d'électronique nous permettra de le comprendre et donc d'y apporter une solution !

4.4.3. Un peu d'électronique

Rappelons qu'un bouton poussoir (dans sa forme la plus courante), réalise un contact momentané quand on appuie dessus, contact qui est rompu dès qu'on relâche le bouton.

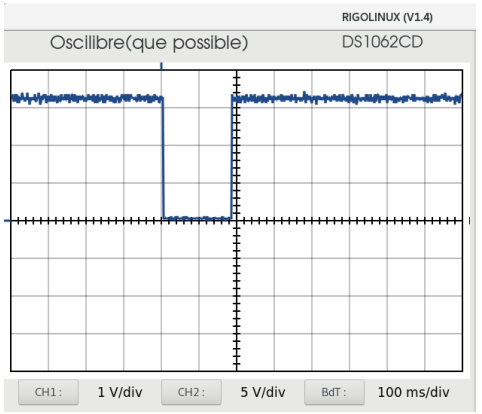
Voici le montage classique pour un bouton poussoir (noté BP, mis en série avec une résistance de 10 kΩ) :



La tension mesurée par le voltmètre (ou par un oscilloscope) sera

- soit égale à 0 V, ce qui donnera un niveau logique 0 appelé état bas
- soit égale à la tension d'alimentation (3,3V sur cet exemple), ce qui donnera un niveau logique 1 appelé état haut.

L'oscillogramme obtenu lors d'un appui bref sur ce bouton :



nous montre que cet appui à une durée supérieure à 100 ms (souvent entre 100 et 200 ms)

(Cet oscillogramme a été obtenu sur un oscilloscope Rigol piloté par le logiciel Rigolinux, logiciel écrit en … Python par l'auteur de ce manuel

et disponible sur le site La Boite à Physique : http://laboiteaphysique.fr/site2/index.php/programmation-et-logiciels/rigolinux)

Par des techniques de hacking que nous ne développerons pas dans ce manuel d'initiation, il est possible d'obtenir la durée d'une portion de code.

Cette boucle :

```
while True:
    display.show(i)
    if i >0 and button_a.is_pressed():
        i = i - 1
    if i<9 and button_b.is_pressed():
        i = i + 1
```

a une durée de l'ordre de 2 ms…

Pendant la durée d'appui sur un bouton (100 à 200 ms pour un appui « bref »), cette boucle est donc parcourue plusieurs dizaines de fois, ce qui explique la variation « instantanée » de la valeur affichée !

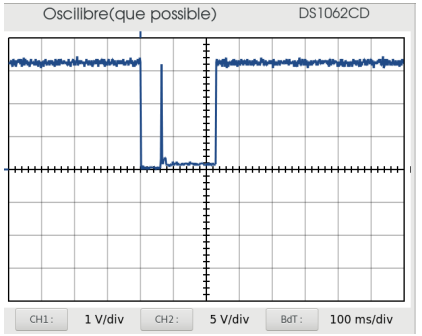
Nous détenons maintenant la solution : il faut ralentir le programme !!! On va donc ajouter une petite temporisation dans la boucle. Prenons 200 ms puisque c'est l'ordre de grandeur d'un appui bref sur un bouton :

```
from microbit import *

i = 5

while True:
    display.show(i)
    if i >0 and button_a.is_pressed():
        i = i - 1
    if i<9 and button_b.is_pressed():
        i = i + 1
    sleep(200)              # temporisation boutons
```

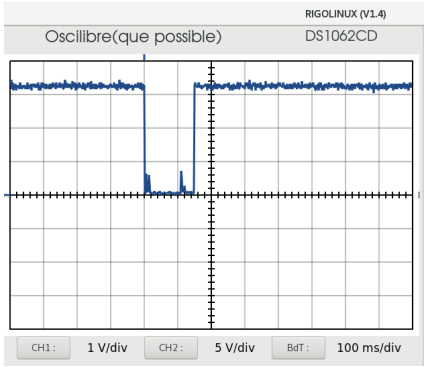
Exécuter ce code et le tester par des appuis brefs ou longs sur l'un ou l'autre des boutons poussoirs.

Rque : un phénomène mécanique peut se produire avec les boutons poussoirs : celui du rebond de contact. Au lieu d'être franc, le contact

donne lieu à des microcoupures qui peuvent être interprétées (selon leur intensité et leur durée) comme de « vrais » appuis sur le bouton poussoirs. En voici des exemples :



Lors de cet appui, le microcontrôleur verra deux fois le bouton pressé



Ici l'impulsion parasite n'est pas assez élevée en valeur de tension pour être interprétée comme un changement d'état : très certainement qu'un seul appui aura été détecté et validé

Exercice 4.4.2_3 : exercice de synthèse

Écrire le code nécessaire au fonctionnement suivant :

- **le programme démarre avec toutes les Leds allumées avec une luminosité égale à 5**
- **un appui sur le bouton A décroît la luminosité de 1**
- **un appui sur le bouton B augmente la luminosité de 1**
- **La luminosité reste toujours comprise entre 0 et 9**

4.4.4. Contrôler le déroulement d'un programme

Reprenons le programme précédent (Ex 4.4.2_3) ... le code fonctionne, mais on veut s'assurer que la luminosité reste bien comprise entre 0 et 9.

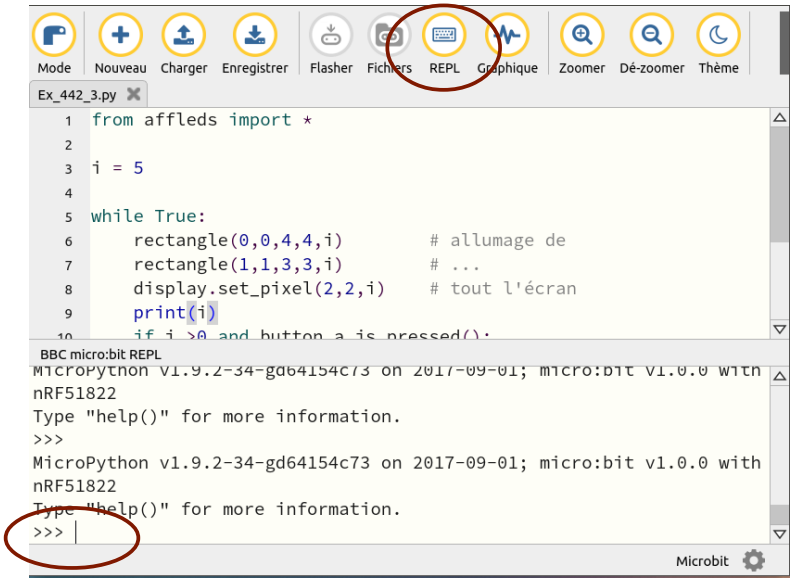
Pour cela, on va utiliser l'**interpréteur interactif REPL (Read-Evaluate-Print Loop)**

Exercice 4.4.4_1 : ajouter dans la boucle infinie l'instruction : print(i)

i étant le nom de la variable utilisée pour la luminosité de l'afficheur dans le reste du programme (à modifier bien sûr en fonction du nom de variable que l'on aura choisi). Si l'on est pas sûr, aller voir la correction (Exercice 4.4.4_1)

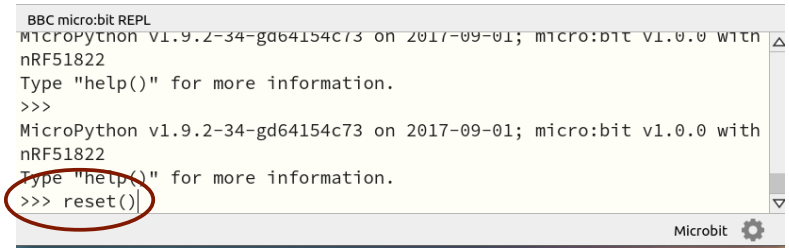
Cette fonction va permettre d'afficher dans cet interpréteur la valeur de la luminosité réglée (i)

- Flasher ce nouveau programme et "jouer" avec les boutons A et B
- Cliquer sur le bouton REPL pour ouvrir l'interpréteur interactif :

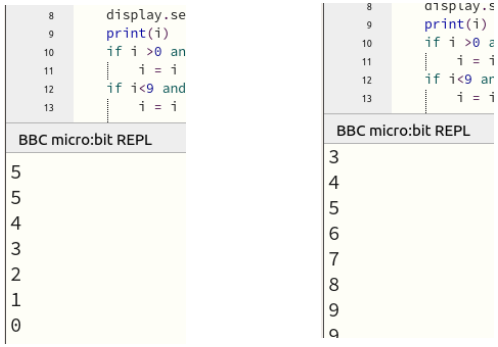


Le logiciel est alors à l'arrêt : le prompt (>>>) attend un ordre. Deux possibilités pour faire démarrer le programme chargé en mémoire :

1. Appuyer sur le bouton de reset situé à l'arrière de la carte
2. ou taper l'instruction reset() dans l'interpréteur interactif et valider l'ordre par la touche Entrée :



A chaque passage dans la boucle, la fonction print(i) nous donne la valeur de la luminosité des Leds.On pourra voir les passages où "i" change, lorsque l'on a appuyé sur les boutons A ou B et ainsi voir que i varie effectivement entre les valeurs 0 et 9 :



Le bouton A fait bien descendre la luminosité jusqu'à la valeur 0 et le bouton B peut la faire remonter jusqu'à 9

Pour arrêter le programme maintenir les touches « CTRL » et « c » appuyées ensemble.

Exercice 4.4.4_2 : Modifier ce programme pour que l'affichage de la luminosité i ne se fasse que lorsqu'un bouton est appuyé.

5 Plus de Python avec l'afficheur

Nous avons utilisé l'afficheur dans diverses situations en utilisant les méthodes suivantes :

- set_pixel() pour allumer une Led. Cette méthode nous a permis également d'apprendre à créer des fonctions pour tracer des lignes ou des quadrilatères.
- show() pour afficher un nombre, un caractère, une suite de caractères ou une image
- scroll(), juste évoquée pour afficher une suite de caractères

Cet afficheur va nous permettre d'approfondir nos connaissances sur le langage Python pour évoquer une notion importante, celle des séquences et découvrir par là même l'instruction for.

Une séquence est une collection ordonnée d'objets. Il y en a trois sortes :

- les strings (ou chaînes de caractères = suite de caractères)
- les tuples
- les listes

Alors que les strings ne contiennent que des suites de caractères, les tuples et les listes peuvent contenir n'importe quel type d'objet. La différence entre les tuples et les listes tient au fait que les listes sont modifiables au cours de l'exécution du programme (on peut venir insérer ou supprimer des éléments).

5.1 Les chaînes de caractères

Une chaîne de caractères est définie entre deux paires d'apostrophes ou deux paires de guillemets. L'un ou l'autre peuvent être utilisés.

Remarque : placer une chaîne de caractères entre deux paires de guillemets permet d'utiliser l'apostrophe à l'intérieur de la chaîne.

Essayons quelques exemples avec la méthode scroll() qui est mieux adaptée que la méthode show() pour afficher une chaîne de caractères.

5.1.1. La méthode scroll()

Il pourra être plus rapide, dans un premier temps, d'essayer les exemples directement dans l'interpréteur interactif comme dans la copie d'écran ci-dessous :

```
BBC micro:bit REPL
>>>
MicroPython v1.9.2-34-gd64154c73 on 2
nRF51822
Type "help()" for more information.
>>> display.scroll("Bonjour !")
>>> texte_fin = 'Au revoir...'
>>> display.scroll(texte_fin)
>>>
```

La méthode scroll() peut prendre quatre paramètres :

scroll(string, delay = 150, wait=True, loop=False, monospace=False)

L'utilisation de ces paramètres est détaillée ci-dessous :

string : la chaîne de caractère à faire défiler. Cela pourra être une chaîne écrite directement entre apostrophes ou entre guillemets (dans l'exemple ci-dessus : "Bonjour !"), ou bien le nom d'une variable représentant une chaîne de caractères (texte_fin dans le deuxième exemple)

Les trois paramètres suivants ont une valeur par défaut. Si on ne les spécifie pas, ce sont ces valeurs par défaut qui seront utilisées) :

delay : permet de régler la vitesse de défilement. A la suite de l'exemple précédent (il faut que la variable texte_fin soit encore en mémoire), essayer :

display.scroll(texte_fin, 500)

pour voir le texte défiler beaucoup plus lentement, le délai étant maintenant passé de la valeur 150 à la valeur 500 (pas d'unité précisée par la documentation)

wait : c'est une grandeur booléenne qui ne peut donc prendre que la valeur True ou la valeur False (valeur par défaut).

- True : le programme va attendre la fin du défilement avant de reprendre le cours du traitement

- False : le programme continue de tourner en tâche de fond pendant le défilement du texte

loop : là aussi une grandeur booléenne définie par défaut à False :

- True : le défilement se fait en boucle
- False : le texte ne défile qu'une seule fois

monospace : dernière variable booléenne qui nous plonge dans les notions liées aux polices de caractères et plus précisément à leur « chasse ». La chasse d'un caractère représente la largeur du caractère plus l'espace qui le sépare du caractère suivant. Comme les caractères n'ont pas tous la même largeur (exemple l et w) on aura deux types de police :

- celles à chasse fixe (reconnaissable par leur nom contenant le terme « Mono ») elles allouent la même largeur à tous les caractères (donc d'avantage de blanc autour du l que du w)
- celle à chasse proportionnelle

  Ci-dessous la même expression en chasse fixe puis en chasse proportionnelle :

  `'Comparer ces deux polices'`

  'Comparer ces deux polices'

Revenons au paramètre monospace :

- True : tous les caractères occupent la même largeur, celle de l'écran (réglage par défaut)
- False : il y a une colonne d'écart entre chaque caractère.

Munis de ces informations, on pourra tester l'effet visuel de ces différents paramètres en les modifiant :

```
display.scroll('Bonjour!',delay = 400, wait=True,loop=False,monospace=False)
```

(le paramètre wait devra être testé dans un programme contenant d'autres lignes de code après celle-ci)

5.1.2. Présenter un résultat numérique

La réalisation de mesures avec un microcontrôleur est une occupation importante en sciences. Une fois la mesure réalisée, il est souvent nécessaire de la retourner à l'utilisateur sous une forme compréhensible (affichage de la valeur, intégration dans un graphe, visualisation à l'aide d'une jauge …). On va voir ici, comment afficher une valeur qui vient d'être mesurée.

Une valeur numérique peut être de type :

- **integer** (un entier positif ou négatif, généralement codé en décimal, mais que l'on peut coder en binaire, en hexadécimal voire en octal)
- **float** (abréviation de floating point, c'est à dire un nombre à virgule flottante)

Exemple : essayer le programme suivant qui affiche un entier :

```
from microbit import *

while True:
    display.scroll(123)
    sleep(1000)
    display.scroll(hex(123))          # conversion en hexadécimal
    sleep(1000)
    display.scroll(bin(123))          # conversion en binaire
    sleep(1000)
```

On verra défiler :

- l'affichage du nombre 123
- sa conversion en hexadécimal (avec la fonction hex()) : 0x7b
- sa conversion en binaire (avec la fonction bin()) : 0b1111011

On peut mélanger du texte et un nombre :

```
display.scroll("L=" + str(53) + " cm")
```

pour afficher une longueur de 53 cm. Deux remarques :

- la chaîne de caractère que l'on affiche provient de la concaténation (= l'assemblage) de trois chaînes réalisée avec **l'opérateur +** :

  - L=
  - la conversion du nombre 53 en une chaîne de caractères avec la fonction **str()** (abréviation de string). L'application de cette fonction transforme le nombre 53 en une suite de deux caractères : « 5 » et « 3 »
  - l'unité cm

On peut aussi avoir récupéré la valeur de la longueur dans une variable. Le code devient alors :

```
longueur = 53
display.scroll("T=" + str(longueur) + "C")
```

Il arrive que la valeur mesurée doive subir un calcul avant d'être affichée, ce qui peut poser un problème de chiffres significatifs. Essayons avec un exemple trivial :

```
from microbit import *
display.scroll(1/3)
```

On demande ici d'afficher le résultat de la division de 1 par 3 ce qui donne 0.333333

Supposons que l'on veuille ne garder que deux chiffres après la virgule, on pourra alors formater le résultat à afficher avec la fonction format qui est utilisée de la façon suivante :

```
'{:.nombre_de_décimalesf}'.format(nombre_à_arrondir)
```

Essayer ce code :

```
from microbit import *
display.scroll('{:.2f}'.format(1/3),loop=True)
```

On peut vouloir aussi afficher le contenu d'une variable et la formater en même temps. Supposons que le résultat d'une mesure de longueur en cm donne 57,463421 , le code pour ne garder que 1 chiffre après la virgule s'écrira :

```
from microbit import *
longueur = 57.463421
```

```
display.scroll('{:.1f}'.format(longueur),loop=True)
```

ce qui donne ici : 57.5 (remarquer l'arrondi qui a été réalisé)

Attention à la façon d'entrer manuellement une donnée numérique : le séparateur décimal utilisé en France est une virgule alors que c' est un point dans les systèmes anglo-saxons : c'est donc un point qu'il faut utiliser !!!

Pour un affichage plus explicite du genre « L=57.5 cm » on concaténera trois chaînes : 'L=' suivie de la chaîne représentant la valeur formatée, elle même suivie de l'unité cm :

```
from microbit import *
longueur = 57.463421
display.scroll('L=' +'{:.1f}'.format(longueur) + 'cm',loop=True)
```

De nombreuses autres options de formatage existent. Le formatage pour affichage d'un nombre à virgule flottante est certainement le plus utile dans le cadre de l'utilisation d'un microcontrôleur, raison pour laquelle nous nous y sommes intéressés de plus près.

5.1.3. Application : mesurer la température ambiante

La carte Micro:Bit ne dispose pas d'un véritable capteur de température. Les concepteurs ont utilisé une astuce à moindre coût : se servir du module de température intégré dans la puce du microcontrôleur. Ces modules de température sont courants sur les processeurs modernes, leur rôle étant de vérifier si la température de la puce ne devient pas trop élevée et donc d'agir en conséquence (mise en action d'un ventilateur par exemple).

Le module température a une résolution de 0,25°C, mais l'instruction MicroPython fournie :

temperature()

retourne un entier donnant en °C la température du microcontrôleur.

On comprendra par là qu'il ne peut s'agir que d'une mesure approximative de la température de l'air ambiant (autour du microcontrôleur) : il ne faut pas que le microprocesseur soit occupé de façon importante (ce qui ferait augmenter sa température… et ne rendrait pas véritablement compte de la température de l'air ambiant).

Ceci dit on est bien là sur un problème de mesure en physique : un thermomètre mesure sa propre température ! A l'utilisateur de faire en sorte qu'il y ait un équilibre (et donc égalité) entre la température du thermomètre et la température du milieu que l'on veut mesurer ... Mettre un thermomètre au soleil pour "mesurer la température qu'il fait au soleil" n'a donc pas de sens !!! … car il suffirait de peindre le thermomètre en noir ... pour avoir plus chaud !

Exercice 5.1.3_1 : Écrire le code permettant d'obtenir sur l'afficheur la valeur de la température sous la forme

T = xx C

Cet affichage se fera en boucle ; on s'assurera que la valeur affichée est bien modifiée, par exemple en soufflant sur le microcontrôleur...

5.2 Les listes et l'instruction for

On a vu au paragraphe 4.4.2 comment afficher une image avec la méthode show. Par exemple :

```
display.show(Image.ARROW_N)
```

pour afficher une flèche verticale : ↑

Pour afficher une succession d'images, il serait tout à fait envisageable de répéter une telle ligne d'instruction autant de fois (moins une !) que l'on a d'images à afficher.

On va voir ici comment afficher une succession d'images en utilisant une autre technique qui fait intervenir une notion importante du langage Python : les listes.

On verra également une autre application des listes lors de la réalisation de séries de mesures avec l'un des capteurs intégrés à la carte Micro:bit ; ces séries de mesures viendront se placer à l'intérieur d'une liste.

5.2.1. Les listes en Python

Entrer le programme suivant dans la mémoire du microcontrôleur :

```
from microbit import *

liste_images = [Image.ARROW_N, Image.ARROW_E,
Image.ARROW_S, Image.ARROW_W]

while True:
    display.show(Image.SQUARE)
    sleep(1000)
    display.show(liste_images[1])
    sleep(1000)
```

On définit ensuite une liste (appelée ici liste_images) contenant quatre éléments faisant référence à des images de flèches différentes dont les orientations sont dans l'ordre :

Nord	Est	Sud	Ouest
ARROW_N	ARROW_E	ARROW_S	ARROW_W
↑	→	↓	←

Dans la définition d'une liste, l'ensemble des éléments se trouve entre crochets [ ] et sont séparés entre eux par une virgule.

Remarque : si l'ensemble des éléments se trouvait entre parenthèses ( ), ce ne serait plus une liste mais un tuple.

On rentre ensuite dans la boucle infinie qui nous rappelle dans un premier temps comment afficher une image prédéfinie dans MicroPython (ici un carré avec Image.SQUARE)

On vient ensuite sélectionner l'affichage d'une des images de la liste avec la ligne d'instruction :

display.show(liste_images[1])

A chaque élément d'une liste est associé un index qui est un entier. Dans cette instruction, on demande d'afficher l'élement de cette liste ayant l'index 1.

Exercice 5.2.1_1 : Exécuter ce programme, observer le résultat. Modifier la valeur de l'index dans l'instruction : display.show(liste_images[1]), puis compléter le tableau suivant :

Forme :	↑	→	↓	←
Nom :	ARROW_N	ARROW_E	ARROW_S	ARROW_W
Index :				

Compléter la phrase suivante :

Le premier élément d'une liste a comme valeur d'index :_____

Exercice 5.2.1_2 : Modifier ce programme pour avoir l'affichage en boucle de chacune de ces images . Chaque image restera une demi-seconde à l'écran.

Exercice 5.2.1_3 : Modifier ce programme pour avoir l'affichage en boucle de toutes les images de flèche prédéfinies dans MicroPython. (Une relecture du paragraphe 4.4.2 pourra être utile). Comme précédemment, chaque image restera une demi-seconde à l'écran.

Exercice 5.2.1_4 : Écrire un programme qui affiche la trotteuse d'une montre à aiguilles.

5.2.2. Parcourir une liste avec l'instruction « for »

Lorsque l'on vient d'un autre langage de programmation, l'instruction « for » fait immédiatement penser à une boucle à compteur. Par exemple en C :

```
for (int i =0 ; i < 10 ; i++)
{
        lignes d'instructions à réaliser 10 fois
}
```

On verra au paragraphe suivant que la boucle à compteur peut être une application de l'instruction « for » en Python, mais dans ce langage cette instruction a été pensée pour avoir une utilisation plus large : l'instruction « for » va permettre de parcourir une séquence, comme par exemple une liste.

Essayer le programme suivant :

```
from microbit import *

liste_images =
[Image.ARROW_N,Image.ARROW_E,Image.ARROW_S,Image.ARROW
_W]
while True:
    for imaj in liste_images:
        display.show(imaj)
        sleep(500)
```

Remarque : on avait obtenu le même résultat visuel avec une boucle while dans l'exercice 5.2.1_2 :

```
while True:
    i = 0
    while i<4:
        display.show(liste_images[i])
        sleep(500)
        i = i + 1
```

Analysons cette nouvelle façon de faire :

```
    for imaj in liste_images:
        display.show(imaj)
        sleep(500)
```

La syntaxe à suivre est la suivante ; elle contient en fait deux instructions (« for » et « in ») :

for *nom_de_variable* **in** *nom_de_la_séquence* **:**

Cette ligne d'instruction se termine par un double point « : ». En Python, cela marque le début d'un bloc d'instructions, bloc qui devra être indenté comme à la suite d'un « while ».

Ce bloc sera parcouru autant de fois qu'il y a d'éléments dans la séquence.

Dans cette ligne d'instruction on commence par définir une variable qui, à chaque passage dans la boucle, contiendra l'un des items de la séquence définie après l'instruction « in »

Dans notre programme :

```
for imaj in liste_images:
```

on définit une variable nommée « imaj » qui prendra ***succesivement et dans cet ordre*** les valeurs suivantes :

Image.ARROW_N,Image.ARROW_E,Image.ARROW_S,Image.ARROW_W

Lors du premier passage dans la boucle, l'instruction display.show(imaj) affichera la flèche « Nord » : ↑, puis la flèche « Est » dans le deuxième passage : → , etc jusqu'au dernier élément de la liste.

Remarque : on n'a pas besoin de compter manuellement le nombre d'éléments présents dans la liste, comme on devait le faire avec la boucle while pour définir le nombre d'itérations.

5.2.3. Une boucle à compteur avec l'instruction « for »

On a vu que le couple d'instructions « for » et « in » permet de parcourir une liste à l'aide d'une variable de même nature que les éléments de la liste. Pour obtenir une boucle à compteur, il suffit de créer une liste d'entiers ; la variable d'itération prendra successivement la valeur de chacun des entiers de la liste.

Tester les programmes suivants dans lesquels différentes listes de nombres eniters ont été créées :

```
from microbit import *

nombres = [0,1,2,3,4]
while True:
    for n in nombres:
```

```
        display.scroll(n)
        sleep(500)
```

```
from microbit import *

nombres = [3,4,5,6]
while True:
    for n in nombres:
        display.scroll(n)
        sleep(500)
```

```
from microbit import *

nombres = [2,4,6,8]
while True:
    for n in nombres:
        display.scroll(n)
        sleep(500)
```

Ces quelques exemples nous auront permis de comprendre comment fonctionne cette boucle à compteur. Pour en faciliter l'écriture, il existe une fonction en Python qui s'appelle range().

Cette fonction peut prendre trois paramètres avec la syntaxe suivante :

range(*debut*,**fin**,*pas*)

Seul le paramètre fin est indispensable.

Exercice 5.2.3_1 : Tester les programmes suivants et indiquer en dessous la séquence d'entiers générée par la fonction range( ) utilisée :

```
from microbit import *

while True:
    for n in range(4):
        display.scroll(n)
```



```
        sleep(500)
```

Dans ce programme, la fonction range(4) a créé la séquence d'entiers suivants :

```
from microbit import *

while True:
    for n in range(-2,4):
        display.scroll(n)
        sleep(500)
```

Dans ce programme, la fonction range(-2,4) a créé la séquence d'entiers suivants :

```
from microbit import *

while True:
    for n in range(-2,4,2):
        display.scroll(n)
        sleep(500)
```

Dans ce programme, la fonction range(-2,4,+2) a créé la séquence d'entiers suivants :

```
from microbit import *

while True:
    for n in range(-5,5,+2):
        display.scroll(n)
        sleep(500)
```

Dans ce programme, la fonction range(-5,5,-2) a créé la séquence d'entiers suivants :

5.2.4. La liste, un conteneur pour les mesures

Nous allons ici se servir d'une liste pour ranger au fur et à mesure qu'elles arrivent les valeurs de « mesures » de la luminosité ambiante. Il n'y a pas de "vrai" capteur de lumière sur la carte Micro:Bit (type LDR ou photodiode par exemple), mais une astuce de la part des concepteurs : utilser en inverse les Leds de l'afficheur pour "mesurer" la luminosité qu'elles perçoivent. Bien sûr les Leds sont alors éteintes !

On a mis mesure entre guillemets car la foncttion méthode utilisée :

display.read_light_level()

ne fait que retourner un entier entre 0 et 255 (donc une "mesure" convertie sur un octet), la valeur augmentant avec la luminosité. Aucune indication n'est fournie sur la linéarité du "capteur" formé par cet ensemble de Leds.

C'est cependant un dispositif pratique (et à coût nul !) pour repérer des variations de luminosité.

Nous allons en profiter pour réinvestir quelques acquis :

- affichage d'un pixel sur l'écran
- utilisation des boutons A et B pour démarrer ou arrêter les mesures
- utilisation avancée de boucles While imbriquées
- la notion de liste en Python
- les fonctions print() et reset() pour interagir dans l'interpréteur REPL (Voir le paragraphe 4.4.4)

et pour explorer quelques notions nouvelles :

- la modification d'une liste en cours de programme avec la méthod append()
- la visualisation des données du REPL sous forme graphique

Entrer le code suivant dans la mémoire du microcontrôleur (attention aux indentations des blocs imbriqués !) :


```
1  from microbit import *
2
3  mesures = []
4  display.set_pixel(2,2,9)
5  boucle_mesure = False
6
7  while True:
8      while boucle_mesure:
9          N = display.read_light_level()
10         mesures.append(N)
11         print(mesures)
12         sleep(1000)
13
14         if button_b.was_pressed():
15             boucle_mesure = False
16             display.set_pixel(2,2,9)
17             print('Arret des mesures')
18
19     if button_a.was_pressed():
20         boucle_mesure = True
21         display.clear()
22         print('Mesures :')
23
```

Analysons ce programme :

- ligne 3 : **mesures = [ ]** , on déclare ici une liste vide appelée "mesures". Cette liste initialement vide va voir sa taille augmenter d'un élément à chaque mesure réalisée. Exemple d'une séquence récupérée dans l'interpréteur interactif :

  [255]
  [255, 29]
  [255, 29, 30]
  [255, 29, 30, 30]

- ligne 4 : **display.set_pixel(2,2,9)** : on allume la Led centrale (bien pratique pour voir à quel moment le programme a démarré ; cela évite par exemple de rentrer dans l'interpréteur interactif alors que le flashage du programme est encore en cours, ce qui provoque alors un dysfonctionnement de la carte.
- ligne 5 : **boucle_mesure = False** on déclare ici une variable booléenne, qui va permettre de parcourir ou pas la boucle de mesures. A False la boucle de mesures ne sera pas parcourue, et donc les mesures seront arrêtées ; à True ce sera l'inverse. Donc au lancement du programme aucune mesure ne sera réalisée. Il faudra faire basculer cette variable à True, ce qui sera fait lorsque

l'on appuiera sur le bouton A (voir les dernières lignes du programme)

- ligne 7 : **while True** : on rentre dans la boucle principale qui tournera tout le temps puisque la condition d'entrée est toujours vraie (True). C'est à l'intérieur de cette boucle principale que va tourner la boucle secondaire dédiée à la mesure et qui démarre à la ligne suivante :
- ligne 8 : **while boucle_mesure** , boucle_mesure qui prendra la valeur True ou False selon les cas (voir explication ligne 5)
- ligne 9 : **N = display.read_light_level()** : on réalise la "mesure" de la luminosité ; le résultat est stocké dans la variable N
- ligne 10 : **mesures.append(N)** : le mot "append" signifie "ajouter" : l'expression **mesures.append(N)** signifie : ajouter le contenu de la variable N à la liste mesures[ ]. (La nouvelle valeur est ajoutée à la fin de la liste)
- ligne 11 : **print(mesures)** : instruction qui va servir dans l'interpréteur interactif. A chaque fois que cette instruction sera lue, l'interpréteur affichera le contenu de la liste mesures[ ]
- ligne 12 : **sleep(1000)** : on fait faire une pause d'une seconde pour ne faire qu'une mesure à chaque seconde environ.
- ligne 14 : on vient ensuite tester si le bouton B *a été* et non pas *est* pressé à l'aide de la méthode "**was_pressed()**" (Essayer le programme en remplaçant les méthodes was_pressed() par is_pressed() pour voir la différence de comportement)
    - ligne 15 : **boucle_mesure = False** : à la prochaine entrée dans la boucle de mesures, la variable boucle_mesure étant False, le programme ira directement au bloc suivant donc en ligne 19, mais avant cela les lignes 16 et 17 seront traitées :
    - ligne 16 : **display.set_pixel(2,2,9)** : on allume la Led centrale ce qui donne à l'utilisateur l'information que l'appui sur le bouton B a bien été pris en compte et que donc les mesures sont arrêtées
    - ligne 17 : **print('Arret des mesures')** : de nouveau une instruction dont le résultat sera visible dans l'interpréteur.
- lignes 19 à 22 : un bloc contenu dans la boucle principal, qui ne sera donc traité que lorsque le programme ne tourne pas à l'intérieur de la boucle de mesures et si le bouton A a été appuyé :

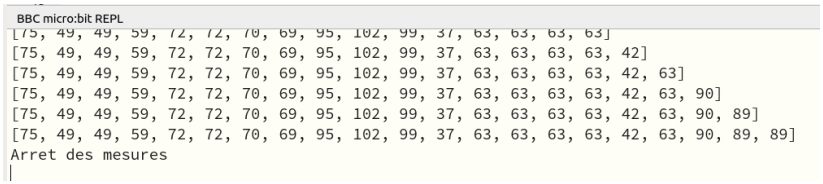
- ligne 20 : **boucle_mesure = True** : à la prochaine entrée dans la boucle de mesures ligne 8, la variable boucle_mesure étant True, le programme entrera effectivement dans cette boucle pour (re)faire des mesures.
- ligne 21 : **display.clear()** : on efface l'afficheur pour indiquer à l'utilisateur que les mesures reprennent.
- ligne 22 : **print('Mesures)** : de nouveau une instruction dont le résultat sera visible dans l'interpréteur.

Utilisation de l'interpréteur interactif :

Lorsque le programme a été chargé en mémoire (avec le bouton "Flasher") :

- lancer l'interpréteur interactif (bouton REPL)
- le programme est arrêté
- taper dans l'interpréteur la commande **reset()** qui va réinitialiser le programme et valider par la touche Entrée du clavier. Si on n'a pas appuyé sur le bouton A alors la Led centrale est restée allumée. Après validation de la commande reset() la Led s'éteint car toutes les fonctions de la carte sont remises à zéro et se rallume dès l'entrée dans le programme (ligne 4) : on sait alors que le programme a démarré. Rien ne se passe dans l'interpréteur jusquà ce que l'on appuie sur le bouton A : là les mesures démarrent (L'interpréteur affiche "Mesures :" conformément à l'instruction ligne 22) et à chaque seconde l'interpréteur affiche le contenu de la liste mesures[ ] qui s'allonge donc de seconde en seconde. On peut alors jouer avec la lumière qui arrive sur le panneau de Leds et voir les valeurs mesurées changer. On pourra arrêter le programme avec le bouton B et le relancer avec le bouton A et ce indéfiniment.

  Un exemple de ce que l'on peut voir dans l'interpréteur ineractif (ici après arrêt des mesures suite à l'appui sur le bouton B)



Visualiser les données dans un graphique

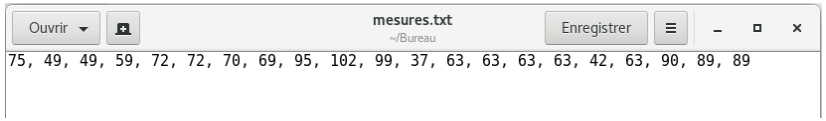
1- Importation manuelle dans un tableur :

Ci-dessous on montre cette possibilité, un peu laborieuse, mais qui permet d'arriver au but recherché néanmoins :

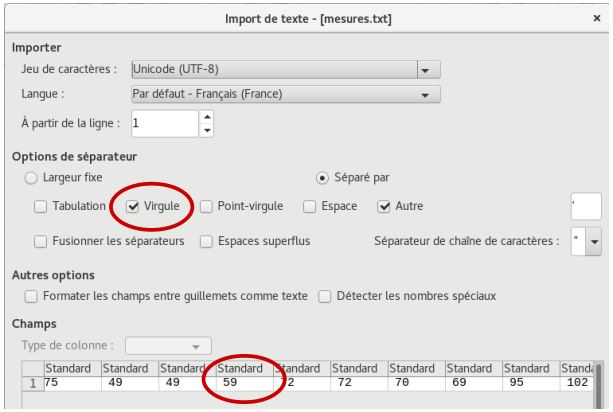
- On sélectionne les données récupérées dans l'interpréteur interactif (éviter de prendre les crochets délimitant la liste) et on les copie dans le presse-papier :

```
BBC micro:bit REPL
[75, 49, 49, 59, 72, 72, 70, 69, 95, 102, 99, 37, 63, 63, 63, 63]
[75, 49, 49, 59, 72, 72, 70, 69, 95, 102, 99, 37, 63, 63, 63, 63, 42]
[75, 49, 49, 59, 72, 72, 70, 69, 95, 102, 99, 37, 63, 63, 63, 63, 42, 63]
[75, 49, 49, 59, 72, 72, 70, 69, 95, 102, 99, 37, 63, 63, 63, 63, 42, 63, 90]
[75, 49, 49, 59, 72, 72, 70, 69, 95, 102, 99, 37, 63, 63, 63, 63, 42, 63, 90, 89]
[75, 49, 49, 59, 72, 72, 70, 69, 95, 102, 99, 37, 63, 63, 63, 63, 42, 63, 90, 89, 89]
Arret des mesures
```

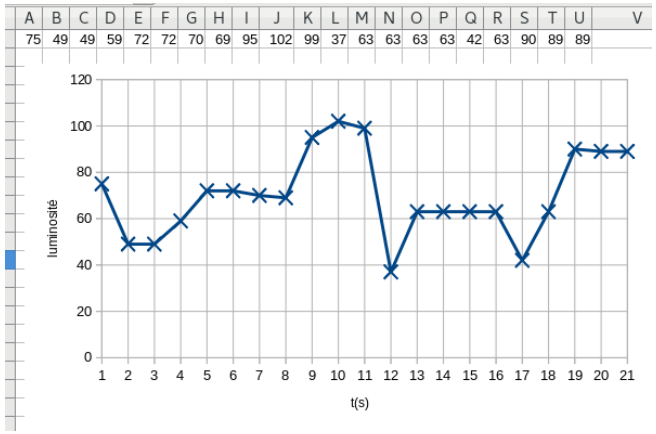
- On les colle dans un éditeur de texte ; on enregistre le fichier (par exemple : mesures.txt)



- On lance le tableur (ici Calc de LibreOffice) et on ouvre le fichier mesures.txt. Dans la boite de dialogue d'importation, on sélectionne les bonnes options pour que chaque valeur de mesure apparaisse dans une cellule :



- Il reste à sélectionner les données pour en faire un graphe (ici on a choisi le mode Line puisque les abscisses sont espacées du même écart d'une seconde).



2- En utilisant l'outil Graphique associé au REPL

Il est nécessaire pour cela de modifier le programme car le graphique ne peut-être obtenu dans ce cas qu'avec des données fournies dans un tuple et non pas dans une liste.

Le tuple est une liste non modifiable (par conséquent, la méthode append n'existe pas pour le tuple). Alors que les élements d'une liste sont donnés entre crochets [ ], ceux du tuple sont donnés entre parenthèses : (()

On doit alors écrire l'expression : **print((display.read_light_level(),))** à l'intérieur de la boucle pour générer à chaque passage un tuple avec un seul élément (la valeur de la mesure). Ci-contre un résultat obtenu dans le REPL

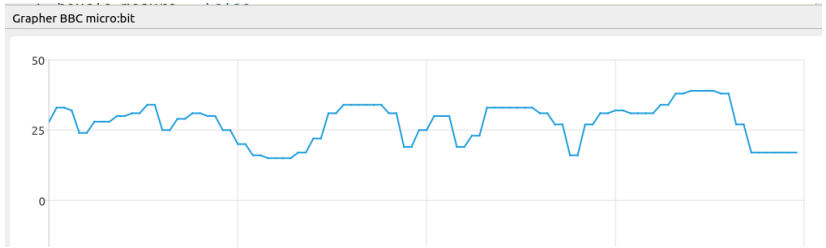
```
BBC micro:bit REPL
(27,)
(27,)
(27,)
(28,)
(27,)
(27,)
(27,)
(27,)
(29,)
(30,)
(30,)
```

Si le tuple est constitué par exemple de 3 éléments, alors le grapheur tracera 3 courbes.

Voici le programme à entrer en mémoire. Une autre différence par rapport au précédent : pour mieux voir les variations, on a réduit ici le delai de boucle à 100 ms :

```
from microbit import *

display.set_pixel(2, 2, 9)
boucle_mesure = False

while True:
    while boucle_mesure:
        print((display.read_light_level(), ))
        sleep(100)
        if button_b.was_pressed():
            boucle_mesure = False
            display.set_pixel(2,2,9)
            print('Arret des mesures')

    if button_a.was_pressed():
        boucle_mesure = True
        display.clear()
        print('Mesures :')
```

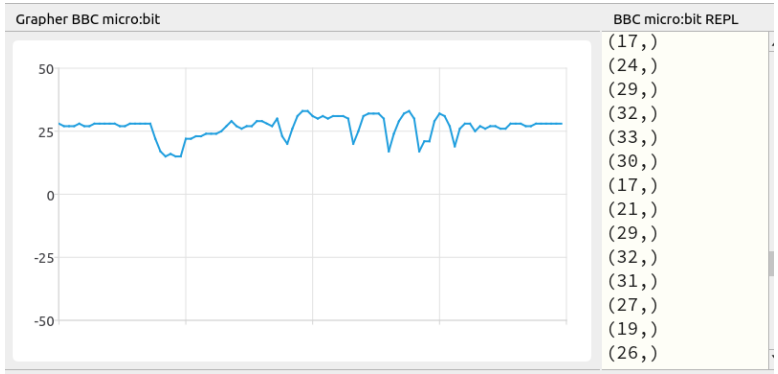
- Charger le programme
- Cliquer sur le bouton "Graphique"
- Appuyer sur le bouton A de la carte Micro:Bit pour lancer les mesures
- Observer le graphique lorsque l'on change la quantité de lumière qui arrive sur "l'afficheur" (ou plutôt le capteur de lumière ici !) :



- Observer ce qui se passe lorsque l'on appuie sur le bouton B de la carte
- Pour conserver le graphique sous forme d'une image on utilisera un logiciel de capture d'écran.

Remarques :

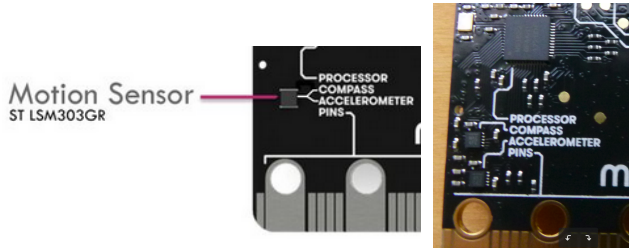
- ***ce grapheur nous semble plutôt limité en possibilités. Son intérêt consiste en la rapidité de mise en œuvre qui peut être bien utile en test. Pour sauvegarder un vrai graphique, on préferera sûrement passer par la méthode précédente qui fournit une liste de données plus faciles à extraire pour les injecter dans un tableur, qui permettra non seulement de visualiser le graphe mais aussi de traiter mathématiquement les données.***
- ***On peut activer à la fois l'interpréteur interactif et le graphique. Cela fractionne la fenêtre de travail en deux parties : l'une avec l'interpréteur et les valeurs numériques qui défilent, l'autre avec le graphe temporel de ces données. On rappelle que l'on peut arrêter le programme dans l'interpréteur interactif avec la combinaison de touches CTRL C :***

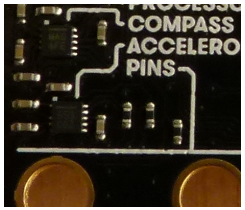


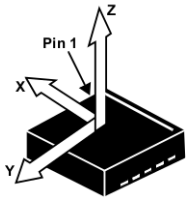
6 Accéléromètre et Magnétomètre

6.1 Présentation

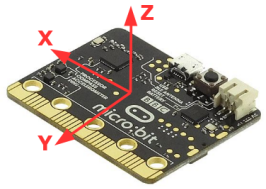
La carte BBC Micro:bit est équipée d'un accéléromètre et d'un magnétomètre tous deux travaillant sur 3 axes (X,Y,Z). Selon la version de la carte, il peut s'agir soit d'un seul circuit incluant les deux capteurs soit de deux capteurs séparés :







En comparant le schéma des axes de l'accéléromètre et l'implantation du composant sur la carte (repérer la barre blanche sur le composant ; les concepteurs ont aussi ajouté un point blanc sur le circuit imprimé pour le repérage de la broche N° 1 du circuit intégré), on peut en déduire l'orientation des axes X,Y,Z pour la carte Micro:Bit.



Attention : en regardant la carte côté afficheur, des axes vont changer de sens...

Sensibilité : l'accéléromètre implanté dispose de plusieurs gammes de mesures, mais sur la carte Micro:Bit, seule la plage **-2g à + 2g** est disponible (g étant bien sûr la valeur de l'accélération de la pesanteur).

Cette mesure se fait avec une *résolution de 10 bits* (1024 valeurs différentes)

Dans cette première approche nous ne détaillerons pas davantage l'aspect technique, car on ne cherchera dans les exemples proposés qu'à détecter la mise en mouvement de la carte.

6.2 Accéder à l'accéléromètre :

En MicroPython, on accède à l'accéléromètre :

- de façon individuelle sur un axe (par exemple l'axe Z) avec la méthode :

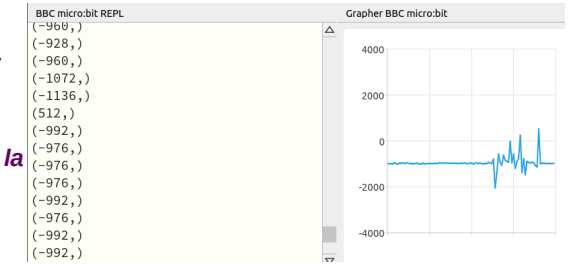
accelerometer.get_z()

qui retourne un entier compris entre +/- 2000 (en 10^{-3}g)

Exemple : pour récupérer la composante sur z de l'accélération dans l'interpréteur interactif , entrer le code suivant :

```
from microbit import *

while True:
    print((accelerometer.get_z(),))
    sleep(100)
```

Remarque : quand on est dans l'interpréteur interactif, le fait de cliquer sur le bouton Graphique, fractionne la fenêtre en deux parties : l'une avec l'interpréteur et les valeurs numériques qui défilent, l'autre le graphe temporel de ces



données. On rappelle que l'on peut arrêter le programme dans l'interpréteur interactif avec la combinaison de touches CTRL C :

- de façon globale sur les trois axes avec la méthode :

accelerometer.get_values()

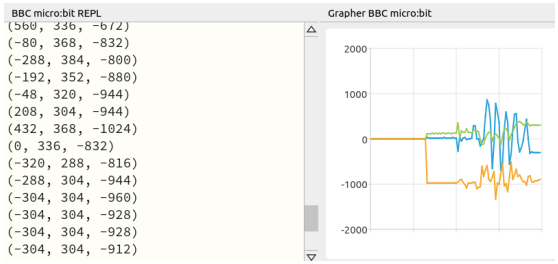
qui va retourner un **tuple** de 3 nombres entiers compris chacun là aussi entre +/- 2000 :

Essayer maintenat l'exemple suivant :

```
from microbit import *

while True:
    print(accelerometer.get_values())
    sleep(100)
```

Attention à la différence d'écriture du code : dans le premier cas, l'instruction accelerometer.get_z() ne retournant qu'un entier, il faut créer le tuple (avec* (accelerometer.get_z(),) *) pour pouvoir visualiser le graphique alors que dans le second cas le tuple est déjà créé par l'instruction accelerometer.get_values()

Ci-dessous la visualisation des tuples (constitués cette fois-ci de trois nombres) et du graphique représentant les composantes sur x,y et z de l'accélération de la carte micro:bit :



Exercice 6.2_1 : écrire le programme qui affiche l'image prédéfinie « ASLEEP » lorsque la carte est immobile, mais l'image « ANGRY » lorsque la carte a été « dérangée ». Cette image restera affichée

pendant 5 secondes... durée pour que la carte décolère et se rendorme !!!

Remarques :

1. ***Il faut bien comprendre que les axes de mesures sont liés à la carte … l'axe Z pourra être un axe vertical si la carte est bien horizontale !***
2. ***De nombreuses applications ludiques peuvent être envisagées avec cet accéléromètre. Citons par exemple la simulation d'un dé, le jeu pierre-ciseau-papier etc. Dans ce type d'application, on aura besoin d'une fonction de génération de nombre aléatoire disponible dans le module de fonctions random dont l'aide se trouve ici :***

https://microbit-micropython.readthedocs.io/en/latest/tutorials/random.html?highlight=random#

6.3 Le magnétomètre

Ce magnétomètre permet d'obtenir en nano tesla la valeur des composantes sur les axes x,y et z de la carte, du champ magnétique détecté.

On va retrouver pour le magnétomètre (« compass » en anglais) le même genre de méthodes que pour l'accéléromètre comme par exemple get_z().

Exercice 6.3_1 : reprendre l'exercice 6.2_1 en remplaçant la détection de mouvement par une détection magnétique : l'approche d'un aimant ou même d'un matériau ferromagnétique déclenchant la colère de la carte.

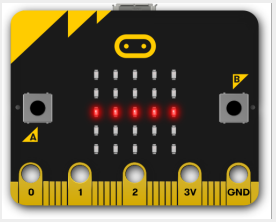
Remarque : l'utilisation précise du magnétomètre passe d'abord pas une étape de calibration qui n'est pas indispensable dans l'exercice proposé. Pour en savoir plus, consulter l'aide :

https://microbit-micropython.readthedocs.io/en/latest/compass.html

7 Solutions aux exercices :

Exercice 4.3.1_1 : afficher le signe -

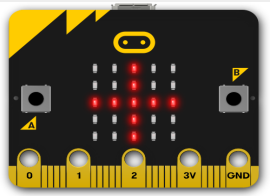
```
from microbit import *

display.set_pixel(0,2,9)
display.set_pixel(1,2,9)
display.set_pixel(2,2,9)
display.set_pixel(3,2,9)
display.set_pixel(4,2,9)
```



Rque : l'ordre dans lequel sont écrites les 5 instructions d'affichage n'a pas d'importance : l'exécution du programme est tellement rapide que l'œil voit les Leds s'allumer en même temps...

Exercice 4.3.2_1 : afficher le signe +

```
from microbit import *
i = 0
while i < 5:
    display.set_pixel(i,2,9)
    display.set_pixel(2,i,9)
    i = i + 1
```



Rque : en exécutant ce programme, les deux lignes d'allumage des Leds donnent la séquence suivante :

```
display.set_pixel(0,2,9)
display.set_pixel(2,0,9)
display.set_pixel(1,2,9)
display.set_pixel(2,1,9)
display.set_pixel(2,2,9)
display.set_pixel(2,2,9)
display.set_pixel(3,2,9)
```



```
display.set_pixel(2,3,9)
display.set_pixel(4,2,9)
display.set_pixel(2,4,9)
```

On constate que l'on demande deux fois au microcontrôleur d'allumer la Led centrale de coordonnées (2,2) ce qui n'est pas un souci : la Led étant déjà allumée la répétition de cette instruction ne change rien !

Exercice 4.3.2_2: Séquence d'affichage du signe + puis du signe - :

```
from microbit import *

# affichage du signe + :
i = 0
while i < 5:
    display.set_pixel(i,2,9)
    display.set_pixel(2,i,9)
    i = i + 1
# réalisation des délais et de l'effaçage :

sleep(1000)                 # 1000 ms = 1 seconde !
display.clear()
sleep(200)                  # 200 ms = 0,2 seconde !

# affichage du signe - :
i = 0
while i < 5:
    display.set_pixel(i,2,9)
    i = i + 1
```

Dans un script Python, le symbole # permet d'écrire des commentaires. A utiliser impérativement dès qu'un programme prend un peu d'ampleur !

Pourquoi est-il indispensable de réécrire la ligne d'instruction i = 0

au dessus de la ligne while i < 5 lorsque l'on veut faire l'affichage du signe - ?

Pour tester une modification dans un programme, par exemple supprimer une ligne pour voir l'effet que cela peut avoir, plutôt que de la supprimer, il est préférable de mettre devant le symbole de commentaire : #. La ligne ne sera pas exécutée et il sera très facile de la récupérer si nécessaire.

Tester le rôle de cette ligne d'instruction i = 0 en la « commentant » (ajout du signe # devant) puis en la « décommentant ».

Exercice 4.3.2_3 : affichage des signes +, - , x et /

```
from microbit import *

while True:
    # affichage du signe + :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        display.set_pixel(2,i,9)
        i = i + 1

    # réalisation du délai et de l'effaçage :
    sleep(1000) #1000ms = 1 seconde !
    display.clear()
    sleep(200)

    # affichage du signe - :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        i = i + 1
    # réalisation du délai et de l'effaçage :
    sleep(1000) #1000ms = 1 seconde !
    display.clear()
    sleep(200)
```



```
    # affichage du signe x :
    i = 0
    while i < 5:
        display.set_pixel(i,i,9)
        display.set_pixel(4-i,i,9)
        i = i + 1

    # réalisation du délai et de l'effaçage :
    sleep(1000) #1000ms = 1 seconde !
    display.clear()
    sleep(200)

    # affichage du signe / :
    i = 0
    while i < 5:
        display.set_pixel(4-i,i,9)
        i = i + 1

    # réalisation du délai et de l'effaçage :
    sleep(1000) #1000ms = 1 seconde !
    display.clear()
    sleep(200)
```

Exercice 4.3.3_1 : écriture d'une fonction avec paramètre :

```
from microbit import *

def delai(dt):
    # réalisation du délai et de l'effaçage :
    sleep(dt)      # dt = délai en millisecondes
    display.clear()
    sleep(200)

while True:

    # affichage du signe + :
```

```
i = 0
while i < 5:
    display.set_pixel(i,2,9)
    display.set_pixel(2,i,9)
    i = i + 1

delai(500)

# affichage du signe - :
i = 0
while i < 5:
    display.set_pixel(i,2,9)
    i = i + 1

delai(1000)

# affichage du signe x :
i = 0
while i < 5:
    display.set_pixel(i,i,9)
    display.set_pixel(4-i,i,9)
    i = i + 1

delai(2000)

# affichage du signe / :
i = 0
while i < 5:
    display.set_pixel(4-i,i,9)
    i = i + 1

delai(4000)
```


Exercice 4.3.3.2 : fonction avec deux paramètres

```
from microbit import *

def delai(dt1, dt2):
    # réalisation du délai et de l'effaçage :
    sleep(dt1)      # dt1 = durée d'affichage en millisecondes
    display.clear()
    sleep(dt2)          # dt2 = durée d'effaçage en millisecondes

while True:

    # affichage du signe + :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        display.set_pixel(2,i,9)
        i = i + 1

    delai(500,200)

    # affichage du signe - :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        i = i + 1

    delai(1000,400)
```

etc..

Rque : il est possible (et parfois souhaitable) de donner une valeur par défaut à tout ou partie des paramètres de la fonction. Dans l'exemple ci-dessus, on pourrait se dire que dans la majorité des cas, la durée pendant laquelle l'écran sera effacé serait de 200 ms, et de préciser la valeur de ce délai dt2 que si l'on veut un autre réglage. Le code devient alors :

```
from microbit import *

def delai(dt1, dt2=200):     # par défaut dt2 vaut 200 ms
```

```
    # réalisation du délai et de l'effaçage :
    sleep(dt1)      # dt1 = durée d'affichage en millisecondes
    display.clear()
    sleep(dt2)          # dt2 = durée d'effaçage en millisecondes

while True:

    # affichage du signe + :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        display.set_pixel(2,i,9)
        i = i + 1

    delai(500)    # dt2 non précisé : il vaudra 200 ms

    # affichage du signe - :
    i = 0
    while i < 5:
        display.set_pixel(i,2,9)
        i = i + 1

    delai(1000,400)    # ici dt2 vaudra 400 ms
```

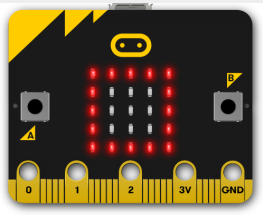
etc...

Exercice 4.3.4_1 : créer une fonction ligne(y)

```
from microbit import *

def ligne(y):
    # allumage d'une ligne de Leds
    # attention : y compris entre 0 et 4
    i = 0
    while i < 5:
        display.set_pixel(i,y,9)
        i = i + 1

ligne(0)
ligne(4)
```


Exercice 4.3.4_2 : programme pour obtenir un grand carré :

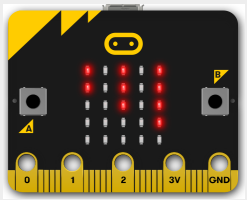
```
from microbit import *

def ligne(y):
    # allumage d'une ligne de Leds
    # attention : y compris entre 0 et 4
    i = 0
    while i < 5:
        display.set_pixel(i,y,9)
        i = i + 1

def colonne(x):
    # allumage d'une colonne de Leds
    # attention : x compris entre 0 et 4
    i = 0
    while i < 5:
        display.set_pixel(x,i,9)
        i = i + 1
while True:
    colonne(0)
    colonne(4)
    ligne(0)
    ligne(4)
```



Exercice 4.3.4_3 : colonne(x1,y1,y2):

```
from microbit import *

def colonne(x1,y1,y2):
    # allumage d'une colonne de Leds
    # x1 et y1 : point de départ haut
    # y2 point d'arrivée
    i = y1
    while i <= y2:
        display.set_pixel(x1,i,9)
        i = i + 1
# exemple d'utilisation :
while True:
```

```
colonne(0,0,1)
colonne(2,0,2)
colonne(4,0,3)
```



Rque : dans le test d'entrée de la boucle conditionnelle :

```
while i <= y2:
```

on regarde si i est inférieur ou égal à y2 (symbole < suivi sans espace du signe =)

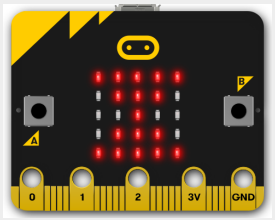
On aurait tout aussi bien pu écrire :

```
while i < y2 + 1 :
```

On peut donc réaliser un calcul dans une instruction de test (ici on fait faire la somme y2 + 1, puis i est comparé à ce résultat)

Exercice 4.3.4_4 : Étant donnée la symétrie du problème, et si on a compris la fonction colonne(x1,y1,y2), il n'est pas difficile d'écrire la définition de ligne(x1,y1,x2) :

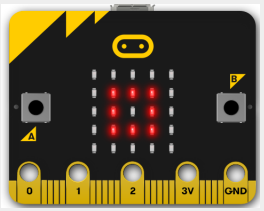
```
from microbit import *

def ligne(x1,y1,x2):
    # allumage d'une ligne de Leds
    # x1 et y1 : point de départ gauche
    # x2 point d'arrivée
    i = x1
    while i <= x2:
        display.set_pixel(i,y1,9)
        i = i + 1
# exemple d'utilisation : sablier
while True:
    ligne(0,4,4)
    ligne(1,3,3)
    display.set_pixel(2,2,9)
    ligne(1,1,3)
    ligne(0,0,4)
```



Exercice 4.3.4_5 :

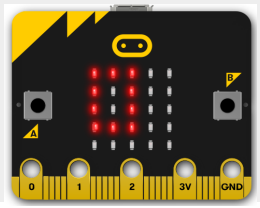
1. **réalisation du carré demandé**

```
from microbit import *

def ligne(x1,y1,x2):
    i = x1
    while i <= x2:
        display.set_pixel(i,y1,9)
        i = i + 1

def colonne(x1,y1,y2):
    i = y1
    while i <= y2:
        display.set_pixel(x1,i,9)
        i = i + 1

while True:
    ligne(1,1,3)
    colonne(1,1,3)
    ligne(1,3,3)
    colonne(3,1,3)
```



pour obtenir le rectangle :

```
while True:
    ligne(0,0,2)
    colonne(0,0,3)
    ligne(0,3,2)
    colonne(2,0,3)
```



Rques :

- *nous avons besoin des fonctions ligne() et colonne() : celles ci doivent être déclarées pour pouvoir être utilisées ensuite.*
- *pour diminuer l'encombrement de ce corrigé, on a retiré les commentaires à l'intérieur des fonctions*

2. réalisation de la fonction rectangle(x1,y1,x2,y2)

Cette définition va s'appuyer sur celles des fonctions ligne() et colonne(). Ces deux fonctions doivent elles aussi être présentes dans le code. Les exemples précédents ont du être une aide pour l'écriture de cette fonction.

```
from microbit import *

def ligne(x1,y1,x2):
    i = x1
    while i <= x2:
        display.set_pixel(i,y1,9)
        i = i + 1

def colonne(x1,y1,y2):
    i = y1
    while i <= y2:
        display.set_pixel(x1,i,9)
        i = i + 1

def rectangle(x1,y1,x2,y2):
    ligne(x1,y1,x2-x1)
    colonne(x1,y1,y2-y1)
    ligne(x1,y2-y1,x2-x1)
    colonne(x2,y1,y2-y1)

while True:
    rectangle(0,0,2,3)    # pour obtenir le rectangle demandé
    #rectangle(1,1,3,3)    # pour obtenir le petit carré central
```

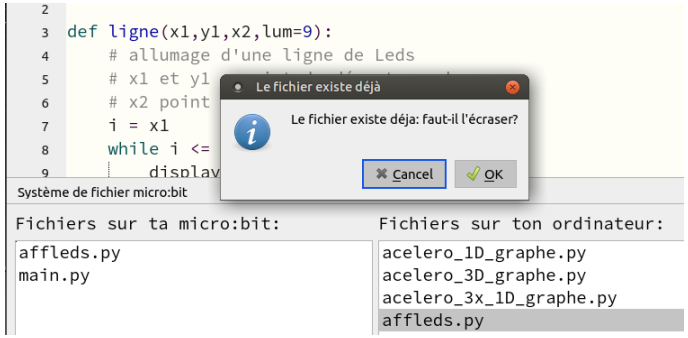
Rque : commenter la ligne rectangle(0,0,2,3) et décommenter la suivante pour changer le motif affiché.

Exercice 4.3.4_5 : Exercice de synthèse

Il faut commencer par réécrire le module de fonctions :

```
from microbit import *
# cette version du module intègre le réglage de luminosité (lum)
# qui peut aller de 0 à 9
def ligne(x1,y1,x2,lum=9):
    # allumage d'une ligne de Leds
    # x1 et y1 : point de départ gauche
    # x2 point d'arrivée
    i = x1
    while i <= x2:
        display.set_pixel(i,y1,lum)
        i = i + 1

def colonne(x1,y1,y2,lum=9):
    # allumage d'une colonne de Leds
    # x1 et y1 : point de départ haut
    # y2 point d'arrivée
    i = y1
    while i <= y2:
        display.set_pixel(x1,i,lum)
        i = i + 1

def rectangle(x1,y1,x2,y2,lum=9):
    # x1, y1 coordonnées du sommet supérieur gauche
    # x2, y2 coordonnées du sommet inférieur droit
    ligne(x1,y1,x2,lum)
    colonne(x1,y1,y2,lum)
    ligne(x1,y2,x2,lum)
    colonne(x2,y1,y2,lum)
```

- puis on le place dans le système de fichiers de la carte :



- on confirme l'écrasement de l'ancien fichier par le nouveau

- on peut tester cela rapidement avec un petit programme comme celui-ci :

```
from affleds import *

while True:
    ligne(0,0,4)    #lum non défini donc = 9
    ligne(0,2,4,6)  #lum = 6
    ligne(0,4,4,3)  #lum = 3
```

qui fera apparaître 3 lignes horizontales de luminosité décroissante

- tester de façon similaire un programme utilisant la fonction colonne
- et enfin un programme utilisant une combinaison de lignes et colonnes :

```
from affleds import *

while True:
    rectangle(0,0,4,4,6)  #luminosité moyenne
    rectangle(1,1,3,3)          #luminosité non définie donc maxi (9)
```

Le programme proposé dans le texte de l'exercice doit montrer l'intégralité de l'écran s'allumer avec une luminosité croissante puis décroissante et ceci en boucle infinie.

Exercice 4.4.2_1 : affichage d'images selon le bouton appuyé

```
from microbit import *

display.show(Image.ASLEEP)

while True:  # création d'une boucle infinie
    if button_a.is_pressed():
        display.show(Image.SAD)
    elif button_b.is_pressed():
        display.show(Image.SMILE)
```


Exercice 4.4.2_2 :

```
from microbit import *

i = 5

while True:
    display.show(i)
    if i >0 and button_a.is_pressed():
        i = i - 1
    if i<9 and button_b.is_pressed():
        i = i + 1
```

On prévoit :

- au démarrage : on lit sur l'afficheur le nombre 5
- lors de l'appui sur le bouton A : le nombre affiché diminuera d'une unité. La valeur minimum sera 0
- lors de l'appui sur le bouton B : le nombre affiché augmentera d'une unité. La valeur maximum sera 9

On constate lors de l'exécution du programme :

- au démarrage : on lit sur l'afficheur le nombre 5
- lors de l'appui sur le bouton A : on passe directement à la valeur 0
- lors de l'appui sur le bouton B : on passe directement à la valeur 9

Alors que le programme semble correctement rédigé, le comportement attendu n'est pas celui observé… Un peu d'électronique va nous aider à démêler l'affaire !

Exercice 4.4.2_3 : Exercice de synthèse

```
from affleds import *

i = 5
```

```
while True:
    rectangle(0,0,4,4,i)       # allumage de
    rectangle(1,1,3,3,i)       # ...
    display.set_pixel(2,2,i)   # tout l'écran

    if i >0 and button_a.is_pressed():
        i = i - 1
    if i<9 and button_b.is_pressed():
        i = i + 1

    sleep(200)  #temporisation boutons poussoirs
```

Exercice 4.4.4_1 : insertion de la fonction print() dans la boucle principale :

```
from affleds import *

i = 5

while True:
    rectangle(0,0,4,4,i)       # allumage de
    rectangle(1,1,3,3,i)       # ...
    display.set_pixel(2,2,i)   # tout l'écran

    print(i)
    if i >0 and button_a.is_pressed():
        i = i - 1
    if i<9 and button_b.is_pressed():
        i = i + 1

    sleep(200)  #temporisation boutons poussoirs
```

Exercice 4.4.4_2 : modification de l'emplacement de la fonction print(i) dans le programme :

```
from affleds import *
```



```
i = 5

while True:
    rectangle(0,0,4,4,i)       # allumage de
    rectangle(1,1,3,3,i)       # ...
    display.set_pixel(2,2,i)   # tout l'écran

    if i >0 and button_a.is_pressed():
        i = i - 1
        print('lum = ',i)
    if i<9 and button_b.is_pressed():
        i = i + 1
        print('lum = ',i)

    sleep(200)  #temporisation boutons poussoirs
```

La fonction print() est insérée dans chaque bloc indenté concernant le traitement des boutons.

Remarquer la syntaxe utilisée ici : print('lum = ', i) qui permet de nommer dans le retour de l'interpréteur la variable affichée :

```
BBC micro:bit REPL
nRF51822
Type "help()" for more i
>>> reset()
lum =  6
lum =  7
lum =  8
lum =  9
```

Cette possibilité d'explicitation est surtout utile lorsque l'on doit afficher les valeurs de plusieurs variables différentes.

Exercice 5.1.3_1 : Écrire le code permettant d'obtenir sur l'afficheur la valeur de la température :

Le listing le plus court pour répondre à la demande de l'exercice :

```
from microbit import *

while True:
    display.scroll('T=' + str(temperature())+ 'C')
```

Rque : le symbole du degré : ° n'est pas disponible...

Si l'on met la valeur de la température dans une variable, cela fonctionne tout aussi bien :

```
from microbit import *

while True:
    temp = temperature()
    display.scroll('T=' + str(temp) + 'C')
```

Par contre si l'on essaye ceci :

```
from microbit import *

while True:
    display.scroll('T=' + str(temperature()) + 'C',loop=True)
```

...l'affichage se fait bien en boucle mais reste constamment avec la valeur de la température saisie au démarrage du programme ! (A essayer pour s'en convaincre...à moins que ce ne soit déjà fait !)

Exercice 5.2.1_1 :

Forme :	↑	→	↓	←
Nom :	ARROW_N	ARROW_E	ARROW_S	ARROW_W
Index :	0	1	2	3

Le premier élément d'une liste a comme valeur d'index : 0

Exercice 5.2.1_2 : A ce stade nous pouvons répondre au problème posé en utilisant une boucle while avec une variable compteur allant de 0 à 3 :

```
from microbit import *

liste_images =
[Image.ARROW_N,Image.ARROW_E,Image.ARROW_S,Image.ARROW
_W]

while True:
    i = 0
    while i<4:
        display.show(liste_images[i])
        sleep(500)
        i = i + 1
```

Rque : la suite du cours va nous montrer une autre façon de traiter ce problème, avec l'utilisation de l'instruction « for ».

Exercice 5.2.1_3 : Les différentes images de flèches prédéfinies sont disponibles sur le site : https://microbit-micropython.readthedocs.io/en/latest/index.html

- `Image.NO`
- `Image.CLOCK12` , `Image.CLOCK11` , `Image.CLOCK10` , `Image.CLOCK9` , `Image.CLOCK8` , `Image.CLOCK7` , `Image.CLOCK6` , `Image.CLOCK5` , `Image.CLOCK4` , `Image.CLOCK3` , `Image.CLOCK2` , `Image.CLOCK1`
- `Image.ARROW_N` , `Image.ARROW_NE` , `Image.ARROW_E` , `Image.ARROW_SE` , `Image.ARROW_S` , `Image.ARROW_SW` , `Image.ARROW_W` , `Image.ARROW_NW`
- `Image.TRIANGLE`

```
from microbit import *

liste_images =
[Image.ARROW_N,Image.ARROW_NE,Image.ARROW_E,Image.ARROW_SE,Image.ARROW_S,Image.ARROW_SW,Image.ARROW_W,
Image.ARROW_NW]
```

```
while True:
    i = 0
    while i<8:
        display.show(liste_images[i])
        sleep(500)
        i = i + 1
```

Exercice 5.2.1_4 : La montre à trotteuse :

```
from microbit import *

liste_images =
[Image.CLOCK12,Image.CLOCK1,Image.CLOCK2,Image.CLOCK3,Ima
ge.CLOCK4,Image.CLOCK5,Image.CLOCK6,Image.CLOCK7,Image.CL
OCK8, Image.CLOCK9, Image.CLOCK10,Image.CLOCK11 ]

while True:
    i = 0
    while i<12:
        display.show(liste_images[i])
        sleep(1000)
        i = i + 1
```

Remarque : à cause de la faible définition de l'écran, le visuel n'est pas fantastique…

Exercice 5.2.3_1 : indiquer en dessous la séquence d'entiers générée par la fonction range( ) utilisée :

```
    for n in range(4):
```

Dans ce programme, la fonction range(4) a créé la séquence d'entiers suivants : 0, 1, 2, 3

```
    for n in range(-2,4):
```

Dans ce programme, la fonction range(-2,4) a créé la séquence d'entiers suivants : -2, -1, 0, 1, 2, 3


```
    for n in range(-2,4,2):
```

Dans ce programme, la fonction range(-2,4,+2) a créé la séquence d'entiers suivants : -2, 0, 2

```
    for n in range(-5,5,+2):
```

Dans ce programme, la fonction range(-5,5,-2) a créé la séquence d'entiers suivants : -5, -3, -1, 1, 3

Conclusion : la fonction range(start, stop, step) crée une séquence d'entiers avec :

- *start* (facultatif) : si cette valeur est précisée ce sera le premier entier de la séquence, sinon la séquence démarre par la valeur 0

- *stop* (obligatoire) : la séquence sera générée jusqu'à cet entier **non inclus.**

- *step* (facultatif) : la différence entre deux entiers consécutifs de la séquence

Exercice 6.2_1 : écrire le programme qui affiche l'image prédéfinie « ASLEEP » lorsque la carte est immobile, mais l'image « ANGRY » lorsque la carte a été « dérangée ». Cette image restera affichée pendant 5 secondes... durée pour que la carte décolère et se rendorme !!!

Ci-dessous une possibilité pour répondre au problème posé :

```
from microbit import *

display.show(Image.ASLEEP)
sleep(5000)

repos = accelerometer.get_z()
#print('repos = ', repos)
```

```
while True:
    etat = accelerometer.get_z()
    #print(etat)
    sleep(100)
    if (etat > (repos + 200) or etat < (repos - 200)):
        display.show(Image.ANGRY)
        sleep(5000)
    display.show(Image.ASLEEP)
```

On est parti du principe suivant : la carte est endormie lorsque elle est horizontale (posée sur une table ou dans le creux ce la main immobile).

On démarre le programme en affichant l'image ASLEEP, on attend 5 secondes avant de prendre la mesure de la composante verticale de l'accélération. Cette valeur est mise dans une variable appelée « repos »

La suite du programme va consister à détecter une variation de cette composante verticale.

On rentre alors dans une boucle infinie dans laquelle on commence par mesurer la composante verticale de l'accélération. Cette valeur est mise dans une variable appelée « etat ».

On vient ensuite tester si la valeur de la variable « etat » a changé de façon *notable*.

Remarquer dans le listing du programme les deux lignes commentées (donc ici non actives) pour afficher dans l'interpréteur interactif la valeur de « repos » et les valeurs prises à chaque tour de boucle par « etat ». Au moment de la mise au point du programme, ces deux lignes étaient actives (donc décommentées) permettant d'estimer quelle variation sur la valeur de la composante il faut garder pour considérer que notre carte a été réveillée. Ci-contre un exemple de relevé.

```
BBC micro:bit REPL
nRF51822
Type "help()" for mo
>>> reset()
repos =  -912
-912
-752
-1088
-960
-976
-2048
```

Une variation de ± 200 nous a semblé raisonnable, d'où la ligne de code :

```
if (etat > (repos + 200) or etat < (repos – 200)):
```


dans laquelle on regarde si la composante a augmenté OU diminué de 200 par rapport à la valeur de repos initiale.

Si l'une de ces deux conditions est réalisée, la carte a été bougée de façon « notable » et on affiche l'image ANGRY pendant 5 secondes.

… on n'oubliera pas le réaffichage de l'image ASLEEP.

Remarque : il n'est pas nécessaire de secouer la carte : le fait de la redresser modifie la composante sur z et suffit donc pour la réveiller !

Exercice 6.3_1 : reprendre l'exercice 6.2_1 en remplaçant la détection de mouvement par une détection magnétique : l'approche d'un aimant ou même d'un matériau ferromagnétique déclenchant la colère de la carte.

On a simplement remplacé les instructions :

accelerometer.get_z()

par :

compass.get_z()

On a choisi une variation de 5000 comme critère de déclenchement (à adapter au type de matériau utilisé (aimant, ou métal) et à la distance de détection souhaitée) sans qu'il ny ait de déclenchement intempestifs

```
from microbit import *

display.show(Image.ASLEEP)
sleep(5000)

repos = compass.get_z()
print('repos = ', repos)
while True:
    etat = compass.get_z()
    print(etat)
    sleep(100)
    if (etat > (repos + 5000) or etat < (repos - 5000)):
        display.show(Image.ANGRY)
        sleep(5000)
    display.show(Image.ASLEEP)
```