# GUIDE MICROPYTHON POUR LA CARTE MICRO:BIT

**Objectifs**

**Connaissances (à savoir)**

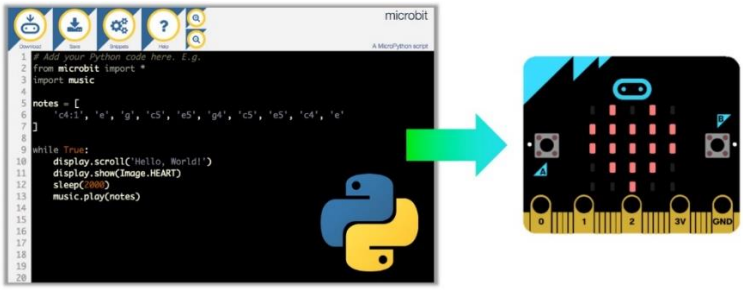
- Savoir retrouver, dans le document ressource, ou ailleurs, les commandes Python utiles à la programmation.

**Capacités (à savoir faire)**

- Être capable de développer des compétences numériques à travers la recherche d'informations et l'exploitation de données et documents numériques ;
- Être capable de programmer la carte micro:bit.
- Être capable de faire un compte-rendu du projet de co-intervention le « robot plastivore ».

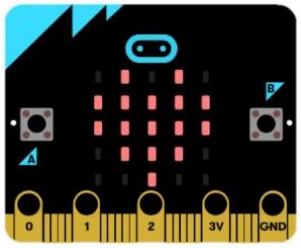
*Python est un langage de programmation utilisé aussi bien par les développeurs débutants que les experts.* ***Python est entièrement basé sur du texte****, ce qui peut paraître intimidant au début, mais avec quelques conseils et un peu de pratique, n'importe qui peut commencer à coder - et ce guide est là pour aider !*

*La version de Python utilisée par le BBC micro:bit est appelée MicroPython. À travers ce document ressource MicroPython, qui décompose chaque fonctionnalités que votre micro:bit peut effectuer à l'aide de Python, vous découvrirez le langage PYTHON.*




## CARTE BBC MICRO:BIT ?

Le BBC micro:bit est un micro-ordinateur de poche programmable qui peut être utilisé pour toutes sortes de créations cool : des robots aux instruments de musique – les possibilités sont très grandes.




On relie la carte micro:bit à l'ordinateur au moyen d'un câble USB. Windows paramètre le pilote lors de la première connexion.

Il faut installer le langage de programmation Python sur l'ordinateur :

## PROGRAMMER LA CARTE MICRO:BIT AVEC L'ÉDITEUR MU

- Une version avancée — et néanmoins très conviviale— de l'éditeur Python de micro:bit, nommée Mu (https://codewith.mu/), peut être téléchargée pour une édition hors-ligne. Cet éditeur présente notamment l'intérêt d'une auto-complétion efficace lors de la saisie du code. Développé en Python, cet éditeur fonctionne sur Windows, MacOS, Linux et Raspberry Pi.

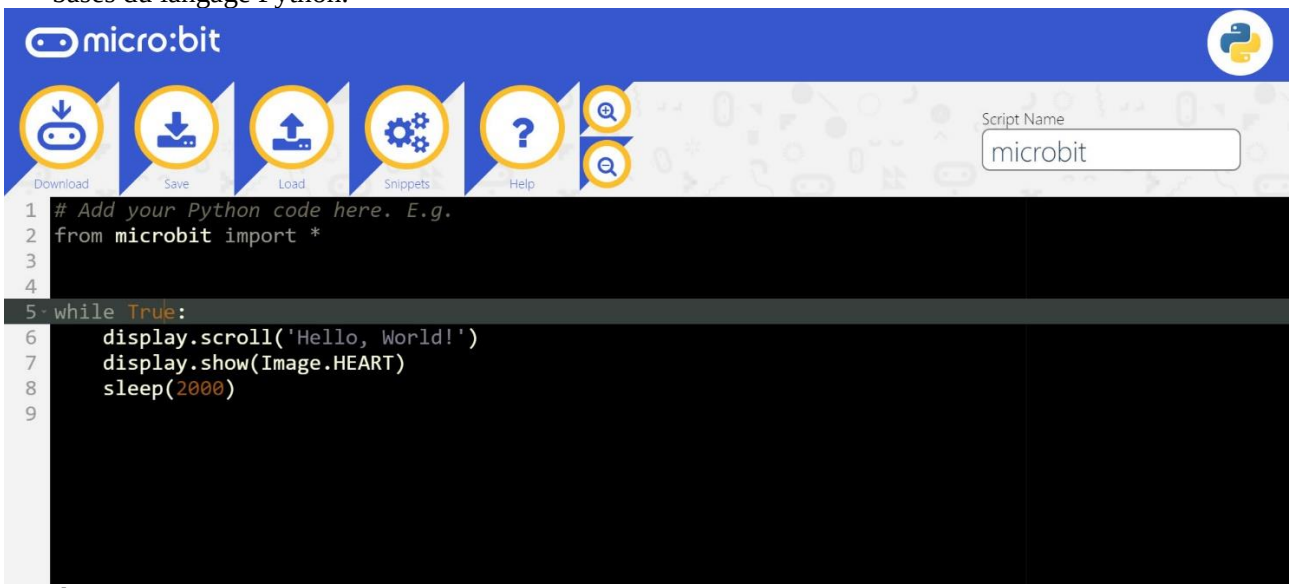

- **Au Lycée**, le raccourci « Mu » est sur le bureau dans le dossier « PYTHON »
- **Chez vous**, vous pouvez installer « **Mu** ». Les instructions pour télécharger et installer Mu se trouvent sur son site Web.
- Une fois que Mu est installé, connectez votre micro: bit à votre ordinateur via un câble USB.

- Écrivez votre script dans la fenêtre d'édition

- **Cliquez sur le bouton "Flash" pour le transférer sur le micro: bit**.

## PROGRAMMER LA CARTE AVEC L'ÉDITEUR PYTHON DU SITE DE LA FONDATION MICRO:BIT

- Cette interface (https://python.microbit.org/v/1.1) fait partie des interfaces proposées sur le site de la fondation micro:bit. Elle s'utilise directement dans un navigateur. Elle nécessite de connaître quelques bases du langage Python.




- **Écrivez votre script** dans la fenêtre d'édition
- Le programme est à sauvegarder (**Save**) au format habituel des programmes Python, .py, et à télécharger (**Download**) sur la carte micro:bit au format .hex.
- **Cliquez sur le bouton "Flash" pour le transférer sur le micro: bit**.
  Si cela ne fonctionne pas, assurez-vous que votre micro: bit apparaît comme un périphérique de stockage USB dans votre explorateur de système de fichiers.

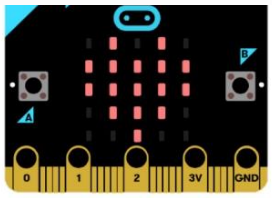
# ACTIVITÉ 1 : HELLO, WORLD! – LES BASES

| SA S'approprier | AN Analyser - Raisonner | RE Réaliser | VA Valider | COM Communiquer |
|---|---|---|---|---|
| A+ | A+ | A+ | A+ | A+ |
| A | A | A | A | A |
| ECA | ECA | ECA | ECA | ECA |
| NA | NA | NA | NA | NA |

A+ : Acquis en autonomie  A : Acquis  ECA : En Cours d'Acquisition  NA : Non Acquis

***Commençons avec les bases - faire défiler "Hello World!" sur ton micro:bit***

La manière traditionnelle de commencer la programmation dans un nouveau langage est de demander à votre ordinateur de dire « Hello, World! ».




C'est facile avec MicroPython:

```
from microbit import *
display.scroll("Hello, World!")
```

Chaque ligne fait quelque chose de spécial. La première ligne:

```
from microbit import *
```

… demande à MicroPython de récupérer tout ce dont il a besoin pour fonctionner avec le micro:bit BBC. Tout cela se trouve dans un module appelé `` microbit ``
***(un module est une bibliothèque de code préexistant)***. Quand tu `` import `` quelque chose, tu dis à MicroPython que tu veux l'utiliser, et `*` est la façon qu'a Python de dire « ***tout*** » . Donc, en français, « je veux pouvoir tout utiliser depuis la bibliothèque de code microbit » s'écrit en Python : `from microbit import *`

La deuxième ligne:

```
display.scroll("Hello, World!")
```

… indique à MicroPython d'utiliser l'affichage pour faire défiler la chaîne de caractères « Hello, World! ». La partie `display` de cette ligne est un *objet* du module `microbit` qui représente l'affichage physique du périphérique (on dit « *objet* » au lieu de « *chose* », « *quoi* » ou « *doodah* »). Nous pouvons dire à l'affichage de faire les choses avec un point `.` suivi de ce qui ressemble à une commande (en fait, c'est quelque chose que nous appelons une ***méthode*** ). Dans ce cas, nous utilisons la méthode `scroll`. Puisque `scroll` doit savoir quels caractères faire défiler sur l'affichage physique, nous les spécifions entre guillemets (( `"` ) entre les parenthèses ( `(` et `)` ). Ce sont les ***arguments***. Ainsi, `display.scroll(« Hello, World! »)` signifie, en français, « Je veux que tu utilises l'écran pour faire défiler le texte "Hello, World!" ». Si une méthode n'a pas besoin d'arguments on utilisant des parenthèses vides comme ceci: `()`.

Copie le code « Hello, World! » dans ton éditeur et flash-le sur le périphérique. Peux-tu trouver comment changer le message? Peux-tu le faire dire bonjour?

Par exemple, je pourrais dire ***« Votre le bateau Plastivore des SMELC, élèves au Lycée de la Méditerranée »***.

Voici un indice ;=)) , tu dois changer l'argument de la méthode de défilement.

AN

***Appel n°1 : Faire vérifier la programmation de la phrase qui va précédente.***

**Avertissement**

Cela peut ne pas fonctionner. :-)

C'est là que les choses amusantes commencent et que MicroPython essaie d'être utile. S'il rencontre une erreur, il fera défiler un message utile sur l'écran du micro-bit. Si c'est le cas, il t'indiquera le numéro de ligne où l'erreur peut être trouvée.

Python s'attend à ce que tu tapes EXACTEMENT la bonne chose. Ainsi, par exemple, `Microbit`, `microbit` et `microBit` sont toutes des choses différentes pour Python. Si MicroPython se plaint à propos d'un `NameError` c'est probablement parce que tu as tapé quelque chose de manière incorrecte. C'est comme la différence entre faire référence à « Nicholas » et « Nicolas ». Ce sont deux personnes différentes mais leurs noms sont très similaires.

Si MicroPython se plaint de `SyntaxError` tu as simplement tapé du code d'une manière que MicroPython ne peut pas comprendre. Vérifie que qu'il ne manque pas de caractères spéciaux comme `"` ou `:`. C'est comme mettre un point au milieu d'une phrase. Il est difficile de comprendre exactement ce que tu veux dire.

Ton microbit peut cesser de répondre: tu ne peux plus lui envoyer un nouveau code ou entrer des commandes dans le REPL. Si cela se produit, essaye de le rallumer. En d'autres termes, débranche le câble USB (et le câble de la batterie s'il est connecté), puis rebranche le câble. Tu devras peut-être également quitter et redémarrer ton éditeur de code.

# ACTIVITÉ 2 : LES IMAGES

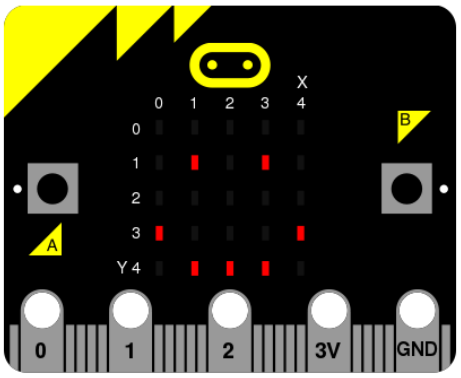
| SA S'approprier | AN Analyser - Raisonner | RE Réaliser | VA Valider | COM Communiquer |
|---|---|---|---|---|
| A+ | A+ | A+ | A+ | A+ |
| A | A | A | A | A |
| ECA | ECA | ECA | ECA | ECA |
| NA | NA | NA | NA | NA |

A+ : Acquis en autonomie  A : Acquis  ECA : En Cours d'Acquisition  NA : Non Acquis

***Prends le contrôle des 25 LEDS de ton micro:bit et découvre les drôles d'images pré-programmées que tu peux utiliser en MicroPython.***

MicroPython est à peu près aussi doué sur le plan artistique que si vous ne disposiez que d'une grille de 5x5 diodes rouges (diodes électroluminescentes - ce qui s'allume à l'avant de l'appareil). MicroPython vous donne beaucoup de contrôle sur l'affichage pour vous permettre de créer toutes sortes d'effets intéressants.
MicroPython est livré avec de nombreuses images intégrées à afficher. Par exemple, pour que le périphérique semble heureux, vous tapez:

```
from microbit import *
display.show(Image.HAPPY)
```




Voici une liste des images intégrées:

| | | | |
|---|---|---|---|
| `Image.HEART` | `Image.HEART_SMALL` | `Image.HAPPY` | `Image.SMILE` |
| `Image.SAD` | `Image.CONFUSED` | `Image.ANGRY` | `Image.ASLEEP` |
| `Image.SURPRISED` | `Image.SILLY` | `Image.FABULOUS` | `Image.MEH` |
| `Image.YES` | `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` | `Image.TRIANGLE_LEFT` | `Image.CHESSBOARD` | `Image.DIAMOND` |
| `Image.DIAMOND_SMALL` | `Image.SQUARE` | `Image.SQUARE_SMALL` | `Image.RABBIT` |
| `Image.COW` | `Image.MUSIC_CROTCHET` | `Image.MUSIC_QUAVER` | `Image.MUSIC_QUAVERS` |
| `Image.PITCHFORK` | `Image.XMAS` | `Image.PACMAN` | `Image.TARGET` |
| `Image.TSHIRT` | `Image.ROLLERSKATE` | `Image.DUCK` | `Image.HOUSE` |
| `Image.TORTOISE` | `Image.BUTTERFLY` | `Image.STICKFIGURE` | `Image.GHOST` |
| `Image.SWORD` | `Image.GIRAFFE` | `Image.SKULL` | `Image.UMBRELLA` |

Il y en a beaucoup! Pourquoi ne pas modifier le code qui fait que le micro: bit a l'air heureux de voir à quoi ressemblent certaines des autres images intégrées? (Il suffit de remplacer `Image.HAPPY` par l'une des images intégrées répertoriées ci-dessus.)

SA

***Appel n°1 : Faire vérifier la programmation de l'image de ton choix.***

# Images de bricolage

Bien sûr, vous voulez créer votre propre image à afficher sur le micro: bit, non?

C'est facile.

Chaque pixel LED de l'affichage physique peut être défini sur dix valeurs. Si un pixel est défini sur `0` (zéro), il est désactivé. Il a littéralement zéro luminosité. Cependant, s'il est réglé sur, `9` il est à son niveau le plus élevé. Les valeurs `1` pour `8` représenter les niveaux de luminosité entre off ( `0` ) et complet on ( `9` ).

Armé de cette information, il est possible de créer une nouvelle image comme celle-ci:

```
from microbit import *

boat = Image("05050:"
             "05050:"
             "05050:"
             "99999:"
             "09990")

display.show(boat)
```

(Lorsqu'il est utilisé, l'appareil doit afficher un voilier à l'ancienne "Blue Peter" avec des mâts plus faibles que la coque du bateau.)

Avez-vous compris comment dessiner une image? Avez-vous remarqué que chaque ligne de l'affichage physique est représentée par une ligne de chiffres se terminant par `:` des `"` guillemets doubles? Chaque nombre spécifie une luminosité. Il y a cinq lignes de cinq chiffres, il est donc possible de spécifier la luminosité individuelle pour chacun des cinq pixels de chacune des cinq lignes de l'affichage physique. Voilà comment créer une nouvelle image.

Facile!

En fait, vous n'avez pas besoin d'écrire cela sur plusieurs lignes. Si vous pensez pouvoir garder une trace de chaque ligne, vous pouvez la récrire comme ceci:

```
boat = Image("05050:05050:05050:99999:09990")
```

SA

***Appel n°2 : Faire vérifier la programmation de votre image de bricolage.***

# Animation

Les images statiques sont amusantes, mais c'est encore plus amusant de les faire bouger. C'est aussi incroyablement simple à faire avec MicroPython ~ utilisez simplement une liste d'images!

Voici une liste de courses:

```
Eggs
Bacon
Tomatoes
```

Voici comment vous représenteriez cette liste en Python:

```
shopping = ["Eggs", "Bacon", "Tomatoes" ]
```

J'ai simplement créé une liste appelée `shopping` et qui contient trois éléments. Python sait que c'est une liste car elle est placée entre crochets ( `[` et `]` ). Les éléments de la liste sont séparés par une virgule ( `,` ) et dans ce cas , les éléments sont trois chaînes de caractères: `"Eggs"` , `"Bacon"` et `"Tomatoes"` . Nous savons que ce sont des chaînes de caractères car ils sont entre guillemets `"` .

Vous pouvez stocker n'importe quoi dans une liste avec Python. Voici une liste de nombres:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

## Remarque

Il n'est pas nécessaire de citer les chiffres car ils représentent une valeur (plutôt qu'une chaîne de caractères). C'est la différence entre `2` (la valeur numérique 2) et `"2"` (le caractère / chiffre représentant le nombre 2). Ne vous inquiétez pas si cela n'a aucun sens pour le moment. Vous allez bientôt vous y habituer.

Il est même possible de stocker différentes sortes de choses dans la même liste:

```
mixed_up_list = ["hello!", 1.234, Image.HAPPY]
```

Remarquez ce dernier élément? C'était une image!

Nous pouvons dire à MicroPython d'animer une liste d'images. Heureusement, nous avons déjà quelques listes d'images intégrées. Elles s'appellent `Image.ALL_CLOCKS` et `Image.ALL_ARROWS` :

```
from microbit import *

display.show(Image.ALL_CLOCKS, loop=True, delay=100)
```

Comme pour une seule image, nous l' `display.show` afficherons sur l'affichage de l'appareil. Cependant, nous disons à MicroPython d'utiliser `Image.ALL_CLOCKS` et elle comprend qu'elle doit afficher chaque image de la liste, l'une après l'autre. Nous disons également à MicroPython de continuer à boucler sur la liste des images (pour que l'animation dure toujours) en disant `loop=True`. De plus, nous lui disons que nous voulons que le délai entre chaque image soit de 100 millisecondes (un dixième de seconde) avec l'argument `delay=100`.

Pouvez-vous travailler comment animer sur la `Image.ALL_ARROWS` liste? Comment éviter de boucler pour toujours (indice: l'inverse de `True` est `False` bien que la valeur par défaut pour `loop` est `False`)? Pouvez-vous changer la vitesse de l'animation?

Enfin, voici comment créer votre propre animation. Dans mon exemple, je vais faire en sorte que mon bateau plonge au bas de l'écran:

```
from microbit import *

boat1 = Image("05050:"
              "05050:"
              "05050:"
              "99999:"
              "09990")

boat2 = Image("00000:"
              "05050:"
              "05050:"
              "05050:"
              "99999")

boat3 = Image("00000:"
              "00000:"
              "05050:"
              "05050:"
              "05050")

boat4 = Image("00000:"
              "00000:"
              "00000:"
              "05050:"
              "05050")
```

```
boat5 = Image("00000:"
              "00000:"
              "00000:"
              "00000:"
              "05050")

boat6 = Image("00000:"
              "00000:"
              "00000:"
              "00000:"
              "00000")

all_boats = [boat1, boat2, boat3, boat4, boat5, boat6]
display.show(all_boats, delay=200)
```

Voici comment fonctionne le code:

- Je crée six `boat` images exactement de la même manière que celle décrite ci-dessus.
- Ensuite, je les mets tous dans une liste que j'appelle `all_boats`.
- Enfin, je demande `display.show` à animer la liste avec un délai de 200 millisecondes.
- Comme je ne suis pas encore installé, `loop=True` le bateau ne coulera qu'une fois (rendant ainsi mon animation scientifiquement exacte). :-)

Que voudriez-vous animer? Pouvez-vous animer des effets spéciaux? Comment feriez-vous une image disparaître puis réapparaître?

AN

***Appel n°3 : Faire vérifier la programmation de l'animation de votre image.***

# ACTIVITÉ 3 : LES BOUTONS

| SA S'approprier | | AN Analyser - Raisonner | | RE Réaliser | | VA Valider | | COM Communiquer | |
|---|---|---|---|---|---|---|---|---|---|
| A+ | | A+ | | A+ | | A+ | | A+ | |
| A | | A | | A | | A | | A | |
| ECA | | ECA | | ECA | | ECA | | ECA | |
| NA | | NA | | NA | | NA | | NA | |

A+ : Acquis en autonomie A : Acquis ECA : En Cours d'Acquisition NA : Non Acquis

***Les boutons A et B sur ton micro:bit sont parmi les nombreuses 'entrées' auxquelles il peut répondre. Apprends comment les utiliser.***

# *Boutons*

Jusqu'à présent, nous avons créé un code qui fait que l'appareil fonctionne. Ceci s'appelle la ***sortie*** . Cependant, nous avons également besoin de l'appareil pour réagir à certaines choses. De telles choses s'appellent des ***entrées*** .

Il est facile de se rappeler: la sortie correspond à ce que l'appareil envoie au monde alors que l'entrée est ce qui entre dans l'appareil à traiter.

Le moyen de saisie le plus évident sur le micro: bit sont ses deux boutons, étiquetés `A` et `B` . D'une manière ou d'une autre, nous avons besoin de MicroPython pour réagir aux pressions sur les boutons.

Ceci est remarquablement simple:

```
from microbit import *


sleep(10000)
display.scroll(str(button_a.get_presses()))
```

Tout ce script ne fait que dormir pendant dix mille millisecondes (c.-à-d. 10 secondes), puis fait défiler le nombre de fois où vous avez appuyé sur le bouton `A` . C'est ça!

Bien que ce soit un script assez inutile, il introduit quelques nouvelles idées intéressantes:

1. La `sleep` *fonction* fera dormir le micro: bit pendant un certain nombre de millisecondes. Si vous voulez une pause dans votre programme, voici comment procéder. Une *fonction* est comme une *méthode* , mais elle n'est pas attachée par un point à un *objet* .
2. Il existe un objet appelé `button_a` qui vous permet d'obtenir le nombre de fois où il a été pressé avec la `get_presses` *méthode* .

Puisque `get_presses` donne une valeur numérique et `display.scroll` n'affiche que des caractères, nous devons convertir la valeur numérique en une chaîne de caractères. Nous faisons cela avec la `str` fonction (abréviation de "chaîne" ~ elle convertit les choses en chaînes de caractères).

La troisième ligne est un peu comme un oignon. Si les parenthèses sont les peaux d'oignons, vous remarquerez que le `display.scroll` contenu contient `str` lui-même `button_a.get_presses` . Python essaie d'abord de trouver la réponse la plus interne avant de passer à la couche suivante. C'est ce qu'on appelle la *nidification* - l'équivalent de codage d'une poupée russe Matriochka.

Imaginons que vous ayez appuyé sur le bouton 10 fois. Voici comment Python détermine ce qui se passe sur la troisième ligne:

Python voit la ligne complète et obtient la valeur de `get_presses` :

```
display.scroll(str(button_a.get_presses()))
```

Maintenant que Python sait combien de touches ont été pressées, il convertit la valeur numérique en une chaîne de caractères:

```
display.scroll(str(10))
```

Enfin, Python sait quoi faire défiler à l'écran:

```
display.scroll("10")
```

Bien que cela puisse sembler beaucoup de travail, MicroPython rend cela extrêmement rapide.

SA

***Appel n°1 : Faire vérifier la programmation du nombre de fois où vous avez appuer sur le bouton A.***

# *Boucles d'événement*

Souvent, votre programme a besoin d'attendre que quelque chose se passe. Pour ce faire, vous le faites en boucle autour d'un morceau de code qui définit comment réagir à certains événements attendus, tels qu'une pression sur un bouton.

Pour créer des boucles en Python, utilisez le `while` mot - clé. Il vérifie si quelque chose est `True`. Si c'est le cas, il exécute un *bloc de code* appelé *corps* de la boucle. Si ce n'est pas le cas, il sort de la boucle (en ignorant le corps) et le reste du programme peut continuer.

Python facilite la définition de blocs de code. Disons que j'ai une liste de tâches écrite sur un bout de papier. Cela ressemble probablement à ceci:

```
Shopping
Fix broken gutter
Mow the lawn
```

Si je souhaite décomposer ma liste de tâches un peu plus loin, je pourrais écrire quelque chose comme ceci:

```
Shopping:
    Eggs
    Bacon
    Tomatoes
Fix broken gutter:
    Borrow ladder from next door
    Find hammer and nails
    Return ladder
Mow the lawn:
    Check lawn around pond for frogs
    Check mower fuel level
```

Il est évident que les tâches principales sont divisées en sous-tâches qui sont en *retrait* sous la tâche principale à laquelle elles sont liées. Donc `Eggs` , `Bacon` et `Tomatoes` sont évidemment liés à `Shopping` . En mettant en retrait des éléments, il est facile de voir en un coup d'œil comment les tâches se rapportent les unes aux autres.

Ceci s'appelle *nidification* . Nous utilisons l'imbrication pour définir des blocs de code comme ceci:

```
from microbit import *

while running_time() < 10000:
    display.show(Image.ASLEEP)

display.show(Image.SURPRISED)
```

La `running_time` fonction retourne le nombre de millisecondes depuis le démarrage de l'appareil.

La ligne vérifie si le temps d'exécution est inférieur à 10000 millisecondes (soit 10 secondes). Si c'est le cas, *et que nous pouvons voir la portée en action* , cela s'affichera . Remarquez comment cela est mis en retrait sous la déclaration, *tout comme dans notre liste de tâches* . `while running_time() < 10000:Image.ASLEEPwhile`

Évidemment, si le temps d'exécution est égal ou supérieur à 10000 millisecondes, l'écran affichera `Image.SURPRISED` . Pourquoi? Parce que la `while` condition sera Faux ( `running_time` n'est plus ). Dans ce cas, la boucle est terminée et le programme continuera après le bloc de code de la boucle. Il semblerait que votre appareil soit endormi pendant 10 secondes avant de se réveiller avec un air surpris. `< 10000while`

Essayez-le!

SA

***Appel n°2 : Faire vérifier la programmation de la boucle d'évenement.***

# *Gérer un événement*

Si nous voulons que MicroPython réagisse aux événements de pression de bouton, nous devons le placer dans une boucle infinie et vérifier si le bouton `is_pressed` .

Une boucle infinie est facile:

```
while True:
    # Do stuff
```

(N'oubliez pas, `while` vérifie si quelque chose `True` doit fonctionner si elle doit exécuter son bloc de code. Comme `True` c'est évidemment `True` pour tous les temps, vous obtenez une boucle infinie!)

Faisons un cyber-animal très simple. C'est toujours triste à moins d'appuyer sur le bouton `A` . Si vous appuyez sur le bouton, `B` il meurt. (Je réalise que ce n'est pas un jeu très agréable, alors peut-être que vous pouvez trouver un moyen de l'améliorer.):

```
from microbit import *

while True:
    if button_a.is_pressed():
        display.show(Image.HAPPY)
    elif button_b.is_pressed():
        break
    else:
        display.show(Image.SAD)

display.clear()
```

Pouvez-vous voir comment on vérifie quels boutons sont appuyés? Nous avons utilisé `if` , `elif` (abréviation de « else if ») et `else` . Celles-ci s'appellent des *conditions* et fonctionnent comme ceci:

```
if something is True:
    # do one thing
elif some other thing is True:
```

```
    # do another thing
else:
    # do yet another thing.
```

Ceci est remarquablement similaire à l'anglais!

La `is_pressed` méthode ne produit que deux résultats: `True` ou `False`. Si vous appuyez sur le bouton, il revient `True`, sinon il retourne `False`. Le code ci-dessus dit, en anglais, "pour toujours et à jamais, si vous appuyez sur le bouton A, affichez un visage heureux, sinon, si vous appuyez sur le bouton B, sortez de la boucle, sinon affichez un visage triste." boucle (arrêter le programme pour toujours et à jamais) avec la `break` déclaration.

À la toute fin, lorsque l'animal domestique virtuel est mort, nous `clear` le présentons.

SA

***Appel n°3 : Faire vérifier la programmation de la boucle d'évenement.***

Pouvez-vous penser à des façons de rendre ce jeu moins tragique? Comment vérifier si les *deux* boutons sont enfoncés? (Indice: Python a `and`, `or` et `not` des opérateurs logiques pour aider à vérifier plusieurs déclarations de vérité (des choses qui produisent l'un `True` ou l' autre ou des `False` résultats).

AN

***Appel n°4 : Faire vérifier la programmation pour rendre ce jeu moins tragique et vérifier si les « deux » boutons sont enfoncés.***