Python

Mise en route

Présentation de Python

Python est un langage de programmation textuel populaire, excellent pour les débutants : il est concis et facile à lire. Il est également utile pour les programmeurs car il s'applique au développement Web et logiciel, ainsi qu'aux applications scientifiques telles que l'analyse de données et l'apprentissage automatique.

La section **Mise en route** présente les bases de l'utilisation de Python avec LEGO® Education SPIKE™ Principal. Elle contient des chapitres où vous allez :

Présentation de Python

1. Apprendre à utiliser l'*éditeur de code* dans l'application LEGO® Education SPIKE™ pour écrire du code Python.

Hello, World!

2. Écrire un message sur la matrice lumineuse du Hub SPIKE Principal.

Commentaires dans Python

3. Découvrir comment les commentaires peuvent vous aider à décrire les programmes provisoires et terminés.

Contrôle des moteurs

4. Définir et démarrer des *fonctions asynchrones* pour contrôler les moteurs.

Variables

5. Contrôler deux moteurs avec des variables *locales* et *globales*.

Le pouvoir du caractère aléatoire

6. Découvrir comment créer des programmes amusants et imprévisibles qui contrôlent la lumière sur le Hub.

Contrôle du capteur

7. Contrôler un moteur à l'aide du capteur de force. Découvrez ensuite comment utiliser la console pour *déboguer* votre programme.

Conditions du capteur

8. Utiliser des *expressions logiques* pour réagir à différentes conditions. Apprenez ensuite à exécuter différentes parties de votre code ensemble pour réagir à plusieurs conditions.

Étapes suivantes

9. Obtenir des suggestions de ressources supplémentaires pour en savoir plus sur l'utilisation de Python avec SPIKE Principal.

Syntaxe de Python

Lors de l'apprentissage d'un langage de programmation de texte, la première étape consiste à comprendre sa *syntaxe*. Cette syntaxe de langage prescrit les règles de rédaction des *instructions* (lignes de code) et la façon d'indiquer les *blocs de code* constitués de plusieurs instructions.

En Python, chaque instruction commence par un niveau de *mise en retrait* et se termine par un *saut de ligne*. La mise en retrait est le nombre d'espaces avant une instruction. Les lignes avec le même nombre d'espaces ont le même *niveau de mise en retrait* et appartiennent au même bloc de code. L'application SPIKE utilise quatre espaces pour chaque niveau de mise en retrait.

Vous écrivez du code dans l'*éditeur de code*, qui dispose de fonctionnalités pour vous aider à l'écrire correctement. Par exemple, lorsque vous démarrez un nouveau bloc de code, comme une *fonction* ou une instruction `if`, l'éditeur met en retrait la ligne suivante avec quatre espaces supplémentaires. En outre, il numérote chaque ligne pour faciliter la navigation dans votre code.

La *coloration syntaxique* dans l'éditeur de code affiche les *commentaires*, les *mots-clés*, le texte et les nombres dans différentes couleurs de manière à ce que le code soit plus facile à lire. Dans le code ci-dessous, le commentaire sur la première ligne est vert, les mots-clés `print`, `if` et `True` sont bleus, le texte `'LEGO'` est magenta et le nombre `123` est orange.

```
# Ceci est un commentaire
print('LEGO')
if True:
    print(123)
```

Le code ci-dessus est un *exemple de programme* que vous trouverez tout au long des chapitres **Mise en route**. Chaque exemple a une icône Copier dans le coin supérieur droit :

Appuyez sur cette icône pour copier l'intégralité de l'exemple de programme. Cliquez ensuite avec le bouton droit de la souris ou appuyez longuement sur l'éditeur de code et sélectionnez Coller dans le menu pour coller le code. Vous pouvez également appuyer sur `CTRL`+`V` sous Windows ou `Command`+`V` sur Mac.

Modules de SPIKE Principal

Pour contrôler le Hub SPIKE Principal, les capteurs et les moteurs, vous aurez besoin des *modules* de SPIKE Principal. Les modules sont utilisés pour organiser le code associé. Il y en a un pour chaque composant SPIKE Principal. Par exemple, le module `motor` contient le code pour contrôler les moteurs. Pour utiliser la fonctionnalité d'un module, *importez*-le d'abord avec l'instruction `import` :

```
import motor
```

Importez les modules dont vous avez besoin une fois, au début de votre programme Python. Consultez la section **Modules de SPIKE Principal** de ce guide de l'utilisateur pour en savoir plus sur les modules et leurs fonctionnalités.

MicroPython

Le Hub SPIKE Principal est un petit ordinateur appelé microcontrôleur, qui a une mémoire et une puissance de traitement limitées. Étant donné que le langage de programmation Python complet utiliserait trop de mémoire, le Hub exécute *MicroPython*, une version hautement optimisée du langage Python, qui peut s'exécuter sur des microcontrôleurs. Les modules de contrôle du Hub SPIKE Principal, des capteurs et des moteurs sont également hautement optimisés grâce à l'utilisation de *types de données* optimisés.

Vous avez vu que l'éditeur de code affiche le texte et les nombres en différentes couleurs, car il s'agit de types de données différents. Python distingue en outre les nombres entiers et les décimales. Les nombres entiers sont également appelés *entiers*, ou type `int`, qui est optimisé dans MicroPython. Les décimales utilisent le type non optimisé `float` de sorte que les modules SPIKE Principal évitent ce type de données. Cela signifie que vous devez vous en tenir à des nombres entiers ou utiliser des *unités* différentes pour décrire les décimales. Pour une demi-seconde, vous pouvez utiliser 500 millisecondes au lieu de 0,5 seconde.

Défi

Arriverez-vous à copier une partie du code d'exemple de ce chapitre et à le coller dans l'éditeur de code ?

Hello, World!

Lors de l'apprentissage d'un nouveau langage de programmation, la tradition veut que l'on crée un programme « Hello, World! ». Vous allez écrire « Hello, World! » sur la matrice lumineuse du Hub SPIKE Principal. Tout d'abord, assurez-vous que votre Hub SPIKE Principal est allumé et connecté à l'application SPIKE. Suivez ensuite ces quatre étapes :

1. Assurez-vous que l'éditeur de code est vide en supprimant tout code existant.
2. Appuyez sur l'icône Copier dans le coin supérieur droit de l'exemple ci-dessous pour copier le code.
3. Faites un clic droit ou appuyez longuement sur l'éditeur de code, puis sélectionnez Coller dans le menu pour coller le code.
4. Appuyez sur le bouton Exécuter pour lancer le programme.

```
from hub import light_matrix

light_matrix.write('Hello, World!')
```

Vous verrez le texte « Hello, World! » défiler sur la matrice lumineuse.

Examinons le code ligne par ligne.

La première ligne importe le module `light_matrix` du module `hub` qui contrôle la matrice lumineuse du Hub. Après avoir importé un module, vous pouvez en utiliser les différentes fonctions.

La dernière ligne *appelle* la *fonction* `write()` du module `light_matrix` pour écrire « Hello, World! » sur la matrice lumineuse.

Définir une fonction

Dans l'exemple précédent, vous avez utilisé la fonction `write()`. Une fonction est un bloc de code qui exécute une tâche lorsque vous l'appelez. Vous définissez une fonction avec le mot-clé `def`, suivi du nom de la fonction, de parenthèses et de deux-points. Le *corps* de la fonction est mis en retrait et contient tout le code qui s'exécute lorsque vous appelez la fonction. Vous appelez une fonction en écrivant le nom de la fonction et des parenthèses. Assurez-vous d'*annuler la mise en retrait* de l'appel de fonction, sinon il fera partie du corps de la fonction.

L'exemple ci-dessous définit la fonction `hello()`, qui écrit « Hello, World! » sur la matrice lumineuse et appelle la fonction une fois. Essayez d'exécuter le code d'exemple. N'oubliez pas de supprimer d'abord tout code existant présent dans l'éditeur de code, avant de copier, coller et exécuter le code.

```
from hub import light_matrix

def hello():
    light_matrix.write('Hello, World!')

hello()
```

Ajouter un paramètre

Dans l'exemple ci-dessus, la fonction `hello()` n'a pas de *paramètres*, elle écrit donc « Hello World » sur la matrice lumineuse chaque fois que vous l'appelez. Pour la rendre plus dynamique, ajoutez un nom de paramètre entre les parenthèses de la définition de fonction. Le bloc de code dans le corps de la fonction peut ensuite utiliser ce paramètre pour faire quelque chose de différent en fonction de sa valeur. On peut citer en exemple la fonction `write()`, qui a un paramètre obligatoire : le texte à écrire sur la matrice lumineuse.

L'exemple ci-dessous ajoute un paramètre `name` à la fonction `hello()`, qui écrit ensuite `'Hello, ' + name + '!'` sur la matrice lumineuse. Remarquez l'*opérateur* `+`, qui vous permet d'ajouter des morceaux de texte ensemble. Le texte est également appelé *chaîne de caractères* ou type `str`. Il est entouré de guillemets droits simples (`'`) ou doubles (`"`), utilisant le même type autour d'une chaîne de texte donnée. La fonction mise à jour `hello()` a un paramètre requis de type `str`, de sorte que vous pouvez *passer* la chaîne de caractères `'World'` en tant qu'*argument* lorsque vous l'appelez pour écrire « Hello, World! » sur la matrice lumineuse.

Essayez d'exécuter le code d'exemple. N'oubliez pas de supprimer tout code existant présent dans l'éditeur de code avant de copier, coller et exécuter votre nouveau code.

```
from hub import light_matrix

def hello(name):
    light_matrix.write('Hello, ' + name + '!')

hello('World')
```

Vous verrez le texte « Hello, World! » défiler à nouveau sur la matrice lumineuse.

Défi

Arriverez-vous à changer le code afin que le Hub *vous* dise bonjour à vous et non pas au monde en général ?

Commentaires dans Python

Il est plus facile d'utiliser du code lorsque vous savez ce qu'il doit faire. Vous pouvez décrire cela en langage courant en ajoutant des commentaires. Les commentaires ne font pas partie du code qui s'exécute sur le Hub. Ils n'influencent donc pas ses fonctionnalités.

Le caractère # marque le début d'un commentaire. Vous placerez généralement un commentaire avant le code qu'il décrit, mais vous pouvez également placer de courts commentaires après une instruction de code.

```
# Ceci est un commentaire
from hub import light_matrix
# Ceci est un autre commentaire
```

Parfois, une partie de votre programme ne se comporte pas comme souhaité. Dans de tels cas, il est utile de *commenter* des parties du code en ajoutant le caractère # au début de la ligne. Ces lignes deviennent alors des commentaires et ne s'exécutent plus dans le cadre de votre programme. Le fait de commenter des parties d'un programme peut vous aider à *déboguer* ou à trouver et corriger le problème. Pour commenter rapidement plusieurs lignes de code, sélectionnez-les, puis appuyez sur `CTRL+/` sous Windows ou `Command+/` sur Mac. Pour retransformer plusieurs commentaires en code, sélectionnez-les et appuyez sur la même combinaison de touches.

```
# La ligne suivante est commentée :
# light_matrix.write('Hello, World!')
```

Vous pouvez également utiliser des commentaires pour décrire votre code avant d'écrire le code de travail. C'est ce qu'on appelle du *pseudo-code*. Cela peut vous aider à décrire dans un langage courant ce que votre programme devrait faire. L'exemple ci-dessous utilise des commentaires comme pseudo-code pour une animation d'yeux clignotants sur la matrice lumineuse.

```
# Affichez un visage souriant avec des yeux sur la matrice lumineuse.
# Attendez un petit peu.
# Affichez un sourire sans yeux sur la matrice lumineuse.
# Attendez un court instant.
# Affichez à nouveau la première image sur la matrice lumineuse.
```

Programme Yeux clignotants

Ce programme affichera un visage avec des yeux clignotants sur la matrice lumineuse du Hub. Copiez le code ci-dessous et collez-le dans l'éditeur de code. Exécutez ensuite le programme. Comme toujours, supprimez tout code existant présent dans l'éditeur de code avant de coller le nouveau code.

Lorsque vous exécutez ce programme, vous verrez que le smiley cligne des yeux après une seconde. Le programme appelle la fonction `show_image()` du module `hub.light_matrix` pour afficher une image sur la matrice lumineuse. Le programme utilise la fonction `sleep_ms()` du module `time` pour ajouter des délais de quelques millisecondes entre les différentes images. Dans le code, chaque commentaire décrit ce que la ligne de code suivante doit faire.

```
import time

from hub import light_matrix

# Affichez un visage souriant sur la matrice lumineuse.
light_matrix.show_image(light_matrix.IMAGE_HAPPY)

# Attendez une seconde.
time.sleep_ms(1000)

# Affichez un sourire sur la matrice lumineuse.
light_matrix.show_image(light_matrix.IMAGE_SMILE)

# Attendez 0,2 seconde.
time.sleep_ms(200)

# Affichez un visage souriant sur la matrice lumineuse.
light_matrix.show_image(light_matrix.IMAGE_HAPPY)
```

« WET » ou « DRY » ?

Bien qu'il soit tentant de commenter chaque ligne de code, le résultat est que vous allez « tout écrire deux fois » (WET, *Write Everything Twice*). Ces commentaires *WET* n'apportent rien aux lecteurs si le code est déjà explicite. Au lieu de cela, suivez le principe *DRY* (*Don't Repeat Yourself*, « ne vous répétez pas »).

Dans l'exemple ci-dessous, les lignes de code qui font clignoter les yeux se trouvent à l'intérieur de la nouvelle fonction `blink()`. Le programme appelle ensuite la fonction trois fois, de manière à ce que les yeux clignotent trois fois. Notez que cette fois, les commentaires ne décrivent que les parties principales du code pour aider les lecteurs à comprendre ce que ce dernier doit faire.

```
import time

from hub import light_matrix

# Cette fonction fait clignoter les yeux.
def blink():
    light_matrix.show_image(light_matrix.IMAGE_HAPPY)
    time.sleep_ms(1000)
    light_matrix.show_image(light_matrix.IMAGE_SMILE)
    time.sleep_ms(200)
    light_matrix.show_image(light_matrix.IMAGE_HAPPY)

# Clignez des yeux trois fois.
blink()
blink()
blink()
```

Défi

Arriverez-vous à changer le code de manière à garder les yeux ouverts plus longtemps chaque fois qu'ils clignotent ?

Contrôle des moteurs

Vous êtes prêt(e) à connecter et à utiliser les moteurs. Connectez un moteur au port A et essayez le programme ci-dessous.

```
import motor
from hub import port

# Faites fonctionner un moteur sur le port A à 360 degrés à la vitesse de 720
degrés par seconde.
motor.run_for_degrees(port.A, 360, 720)
```

Vous devriez voir le moteur tourner à 360 degrés (une rotation complète) à la vitesse de 720 degrés (deux rotations) par seconde.

Examinons le code ligne par ligne.

La première ligne importe le module `motor` qui contrôle les moteurs.

La deuxième ligne importe `port` du module `hub`, qui contient la valeur de chaque port. Vous pouvez écrire `port.A` pour le port A, `port.B` pour le port B, etc. pour spécifier le(s) port(s) souhaité(s).

La dernière ligne appelle la fonction `run_for_degrees()` avec trois *arguments* :

1. Le premier paramètre spécifie le moteur à faire fonctionner, à l'aide de la valeur du port.
2. Le deuxième paramètre spécifie le nombre de degrés à exécuter.
3. Le troisième paramètre spécifie la vitesse à laquelle faire fonctionner le moteur, en degrés par seconde.

Plusieurs moteurs

À présent, connectez un deuxième moteur au port B et essayez le programme ci-dessous.

```
import motor
from hub import port

# Faites fonctionner deux moteurs sur les ports A et B à 360 degrés à 720 degrés
par seconde.
# Les moteurs tournent en même temps.
motor.run_for_degrees(port.A, 360, 720)
motor.run_for_degrees(port.B, 360, 720)
```

Notez que les deux moteurs tournent à 360 degrés (une rotation) à la vitesse de 720 degrés par seconde. Ils démarrent et s'arrêtent en même temps. Étant donné que les deux instructions relatives aux moteurs sont sur des lignes distinctes, vous pouvez vous attendre à ce qu'elles s'exécutent l'une après l'autre. Cependant, elles s'exécutent en même temps car `run_for_degrees()` est une fonction *attendable* (que l'on peut attendre). Cela signifie que vous *pouvez* attendre qu'elle se termine, mais que vous n'êtes pas obligé(e) de le faire. Par défaut, le programme passe immédiatement à la ligne de code suivante tandis que le code attendable s'exécute jusqu'à la fin en arrière-plan. Cela permet d'exécuter plusieurs commandes en même temps.

Boucle d'exécution, Asynchrone et Await

Pour utiliser efficacement du code attendable (que l'on peut attendre) avec la possibilité d'exécuter des commandes de manière simultanée ou séquentielle, vous devez exécuter votre code dans une *fonction asynchrone* à l'aide d'une *boucle d'exécution*. Le module `runloop` contrôle la boucle d'exécution sur le Hub et vous permet d'exécuter des fonctions asynchrones avec sa fonction `run()`. Une fonction asynchrone, également appelée *coroutine*, est une fonction attendable qui utilise le mot-clé `async` avant la définition de la fonction. La convention est de nommer la coroutine contenant votre programme principal `main()`. Le code ci-dessous montre la structure générale d'un programme utilisant une boucle d'exécution.

```
import runloop

async def main():
    # Écrivez votre programme ici.

runloop.run(main())
```

Dans le corps d'une coroutine, vous pouvez utiliser le mot-clé `await` avant d'appeler une commande attendable. Cela met en pause la coroutine jusqu'à la fin de la commande. Sans le mot-clé, le programme passe immédiatement à la ligne de code suivante dans la coroutine. Vous pouvez toujours utiliser du code normal (non attendable - que l'on ne peut pas attendre) dans une coroutine. Cependant, cela mettra toujours en pause ou *bloquera* l'ensemble du programme jusqu'à ce que la commande soit terminée.

Le programme ci-dessous définit la coroutine `main()`, qui utilise le mot-clé `await` avant les deux appels de fonction `run_for_degrees()`. Il utilise la fonction `run()` du module `runloop` pour exécuter la coroutine `main()` sur la dernière ligne de code.

```
import motor
import runloop
from hub import port

async def main():
    # Faites fonctionner deux moteurs sur les ports A et B à 360 degrés à 720
degrés par seconde.
    # Les moteurs tournent l'un après l'autre.
    await motor.run_for_degrees(port.A, 360, 720)
    await motor.run_for_degrees(port.B, 360, 720)

runloop.run(main())
```

Essayez l'exemple de code. Vous devriez voir que les deux moteurs tournent à 360 degrés (une rotation) à la vitesse de 720 degrés par seconde, un après l'autre.

Défi

Arrivez-vous à modifier le code pour faire à nouveau fonctionner les deux moteurs en même temps ?

Variables

Parfois, vous vous retrouvez à écrire le même nombre encore et encore. Par exemple, les commandes du moteur du chapitre précédent ont fonctionné avec le même nombre de degrés, à la même vitesse, à chaque fois. Dans de tels cas, l'utilisation de variables facilite la modification de plusieurs commandes.

Vous créez une variable en écrivant son nom, suivi d'un signe `=` unique et de la valeur initiale de cette variable. Si vous souhaitez modifier la valeur d'une variable existante, vous devez utiliser exactement le même format pour lui *attribuer* une nouvelle valeur.

Connectez les moteurs aux ports A et B et essayez le programme ci-dessous.

```
import motor
import runloop
from hub import port

async def main():
    # Créez une variable `velocity` avec une valeur de 720.
    velocity = 720

    # Faites fonctionner deux moteurs sur les ports A et B à 360 degrés.
    # Utilisez la valeur de la variable `velocity` pour la vitesse du moteur.
    await motor.run_for_degrees(port.A, 360, velocity)
    await motor.run_for_degrees(port.B, 360, velocity)

runloop.run(main())
```

Comme dans le chapitre précédent, vous verrez les deux moteurs tourner à 360 degrés (une rotation) à la vitesse de 720 degrés par seconde, un après l'autre. L'exemple ici crée une variable `velocity` et l'utilise dans les appels de la fonction `run_for_degrees()`. Étant donné que nous avons utilisé une variable, il est facile de changer la vitesse du moteur pour toutes les commandes du moteur. Essayez de modifier la valeur de la variable `velocity` et exécutez à nouveau le programme.

Portée de la variable

Il est important de comprendre l'importance de l'endroit *où* une variable est créée. Lorsque vous créez une variable à l'intérieur d'une fonction, elle n'est disponible que pour cette fonction. C'est ce qu'on appelle une variable *locale*. Si vous souhaitez utiliser une variable entre différentes fonctions de votre programme, vous devez créer la variable en dehors des fonctions, par exemple sous vos instructions `import`. C'est ce qu'on appelle une variable *globale*.

```
import motor
import runloop
from hub import port

# Créez une variable globale `velocity` avec une valeur de 720.
velocity = 720

async def main():
```

```
    # Créez une variable locale `degrees` avec une valeur de 360.
    degrees = 360

    # Faites fonctionner deux moteurs sur les ports A et B.
    # Utilisez la valeur de la variable `degrees` pour le nombre de degrés.
    # Utilisez la valeur de la variable `velocity` pour la vitesse du moteur.
    await motor.run_for_degrees(port.A, degrees, velocity)
    await motor.run_for_degrees(port.B, degrees, velocity)

runloop.run(main())
```

Encore une fois, vous verrez les deux moteurs tourner à 360 degrés (une rotation) à la vitesse de 720 degrés par seconde, un après l'autre. Cette fois, la variable `velocity` a une *portée* globale et une nouvelle variable `degrees` a une portée locale. Vous pouvez utiliser la variable globale `velocity` à la fois à l'intérieur et à l'extérieur de la fonction `main()`, mais vous ne pouvez utiliser la variable locale `degrees` qu'à l'intérieur de la fonction `main()` où elle est définie.

Il peut être tentant de définir toutes vos variables en haut de votre programme afin qu'elles aient une portée globale, car ainsi vous pourrez les utiliser facilement dans l'ensemble de votre programme. Cependant, cela signifie également que la valeur de ces variables peut être modifiée de *n'importe où* dans votre programme, avec des effets secondaires indésirables. Au lieu de cela, *limitez étroitement* vos variables, de sorte que seules les parties de votre programme qui doivent les utiliser et les modifier y aient accès.

Variables dans les boucles

En Python, le moyen le plus simple de répéter du code un certain nombre de fois est d'utiliser une boucle `for` avec la fonction intégrée `range()`. Par exemple, pour répéter quelque chose quatre fois, vous écrivez `for i in range(4):` suivi du code que vous souhaitez exécuter quatre fois. Vous pouvez considérer `range(4)` comme le *tuple* `(0, 1, 2, 3)`. Les tuples et les *listes* telles que `[1, 2, 3]` sont *itérables*. La boucle `for` prend un itérable et *forme une boucle* sur ses valeurs jusqu'à ce qu'elle atteigne la fin.

Lorsqu'une boucle `for` *itère* sur un tuple ou une liste, elle modifie la valeur d'une variable locale à chaque itération. Jusqu'à présent, vous avez explicitement créé des variables et leur avez attribué une valeur à l'aide du signe `=`. Dans une boucle `for`, le nom de la variable locale est défini après le mot-clé `for`, dans ce cas `i`. Chaque fois que la boucle s'exécute, la valeur de la variable locale `i` change. Ce sera `0` pour la première exécution de la boucle et `3` lors de sa dernière exécution, correspondant aux valeurs de `(0, 1, 2, 3)`.

L'exemple suivant utilise une boucle `for` pour modifier quatre fois la variable globale `velocity` afin de faire fonctionner le moteur sur le port A à une vitesse différente à chaque fois. Pour activer la modification de la variable globale `velocity` dans le *contexte local* de la fonction `main()`, vous devez utiliser le mot-clé `global` avant `velocity` au début du corps de la fonction.

```
import motor
import runloop
from hub import port

# Créez une variable globale `velocity` avec une valeur de 450.
```

```
velocity = 450

async def main():
    # Utilisez le mot-clé `global` pour activer la modification de `velocity`
ici.
    global velocity

    # Créez une variable locale `degrees` avec une valeur de 360.
    degrees = 360

    # La boucle `for` crée une variable locale `i` et se répète 4 fois.
    # Les valeurs de la variable `i` sont 0, 1, 2 et 3.
    for i in range(4):
        # Modifiez la variable globale `velocity` en ajoutant `i`*90 à chaque
fois.
        # Les valeurs de la variable `velocity` sont 450, 540, 720 et 990.
        velocity = velocity + i*90
        await motor.run_for_degrees(port.A, degrees, velocity)

    # La valeur de la variable `velocity` en dehors de la boucle `for` est 990.
    await motor.run_for_degrees(port.B, degrees, velocity)

runloop.run(main())
```

Exécutez le code d’exemple. Vous verrez le moteur sur le port A tourner à 360 degrés quatre fois, à quatre vitesses différentes, plus rapidement à chaque fois. La dernière fois, le moteur sur le port B tourne à 360 degrés une fois à la vitesse de 990 degrés par seconde.

Défi

Arriverez-vous à modifier le code pour que le moteur du port A fonctionne à un nombre de degrés différent à chaque fois ?

Le pouvoir du caractère aléatoire

Parfois, le meilleur programme est celui qui est imprévisible. Lorsque vous ignorez ce qu’un programme fera ensuite, il paraît plus dynamique. Pour obtenir ce résultat, ajoutez un peu de caractère aléatoire.

Le programme ci-dessous réglera le voyant du bouton d’alimentation du Hub SPIKE Principal sur dix couleurs différentes, avec un délai aléatoire entre chaque changement de couleur.

```
import random
import time
from hub import light

for color in range(11):
    # Réglez la lumière sur la couleur actuelle.
    light.color(light.POWER, color)

    # Maintenez la lumière allumée pendant 0,5 à 1,5 seconde.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)
```

Chaque couleur est représentée par un nombre différent. La boucle `for` itère sur `range(11)` et attribue sa valeur à la variable `color`. Ce sera `0` (noir) qui éteindra la lumière la première fois que la boucle fonctionnera et `10` (blanc) lors de la dernière itération. Notez que ce programme importe le module `random`, qui contient plusieurs fonctions permettant d’ajouter du caractère aléatoire.

Cet exemple utilise la fonction `randint()`, avec une valeur `start` de 500 et une valeur `stop` de 1500. Avec ces arguments, la fonction *renvoie* un nombre compris entre 500 et 1500 pour ajouter une certaine variante au temps de veille. Cependant, les couleurs s'allumeront toujours dans le même ordre, même si vous exécutez le programme plusieurs fois. Heureusement, le module `random` a d'autres fonctions pour ajouter encore plus de caractère aléatoire au programme.

Boucle infinie

Vous pouvez également utiliser une boucle `while` pour répéter quelque chose à l'infini plutôt qu'un nombre spécifique de fois. En Python, le moyen le plus simple de créer une telle boucle est d'écrire `while True:` suivi du code que vous souhaitez exécuter à l'infini. L'exemple suivant utilise une boucle `while True` pour exécuter un petit spectacle disco à l'infini ou jusqu'à ce que vous arrêtiez le programme.

```
import random
import time
from hub import light

while True:
    # Générez un nombre aléatoire compris entre 1 et 9.
    random_color = random.randint(1, 9)

    # Réglez la lumière sur la couleur aléatoire.
    light.color(light.POWER, random_color)

    # Maintenez la lumière allumée pendant 0,5 à 1,5 seconde.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)
```

Notez que le bouton d'alimentation s'allume dans des couleurs aléatoires, avec un délai variable entre chaque changement de couleur. L'exemple utilise à nouveau la fonction `randint()` pour générer un nombre compris entre 1 et 9 (tous deux inclus), ce qui correspond aux différents nombres désignant les couleurs, à l'exclusion du noir (`0`) et du blanc (`10`).

Listes et constantes

Si vous souhaitez que les lumières affichées n'incluent que certaines couleurs, vous pouvez les mettre dans une *liste* puis choisir une couleur aléatoire à partir de là. Pour créer une nouvelle liste, vous procédez comme pour une variable, en écrivant d'abord le nom de la liste, puis le signe `=`, et enfin les valeurs entre crochets, séparées par des virgules. Pour une liste simple avec au moins deux *éléments*, écrivez `my_list = [1, 2]`. Vous pouvez ajouter autant de valeurs que vous le souhaitez.

Comme vous l'avez vu dans les exemples précédents, chaque couleur est représentée par un nombre différent. Vous utilisez ce nombre pour définir la couleur de la lumière. Par exemple, le nombre `9` réglera la lumière sur le rouge. Cependant, l'utilisation de nombres pour désigner les couleurs peut rendre plus difficile pour les autres lecteurs de savoir ce que va faire votre code. Vous pouvez ajouter des commentaires pour décrire chaque valeur, mais il peut être préférable de créer des variables pour chaque couleur. Le module `color` a une variable `RED`, vous pouvez donc écrire `color.RED` à la place du nombre `9` dans votre code. (Une variable écrite en lettres majuscules est une *constante*, ce qui signifie que vous ne devez pas la modifier.)

L’exemple ci-dessous importe le module `color` et utilise certaines des constantes de couleur pour créer la liste `colors`. Cette fois, la fonction `randint()` détermine le nombre de fois où la boucle `for` s’exécute. La lumière devient blanche à la fin de ce petit spectacle lumineux aléatoire.

```
import random
import time
import color
from hub import light

# Créez une liste avec différentes couleurs de lumière.
colors = [color.RED, color.GREEN, color.BLUE, color.YELLOW]

# Modifiez la lumière entre cinq et dix fois.
times = random.randint(5, 10)

for i in range(times):
    # Choisissez une couleur aléatoire dans la liste des couleurs.
    random_color = random.choice(colors)

    # Réglez la lumière sur la couleur aléatoire.
    light.color(light.POWER, random_color)

    # Maintenez la lumière allumée pendant 0,5 à 1,5 seconde.
    sleep_time = random.randint(500, 1500)
    time.sleep_ms(sleep_time)

# Réglez la lumière sur le blanc.
light.color(light.POWER, color.WHITE)
```

Vous verrez le voyant du bouton d’alimentation passer à une couleur aléatoire de la liste, un nombre aléatoire de fois, avec un délai différent entre chaque changement de couleur. L’exemple utilise la fonction `choice()` pour choisir une couleur au hasard dans la liste `colors`.

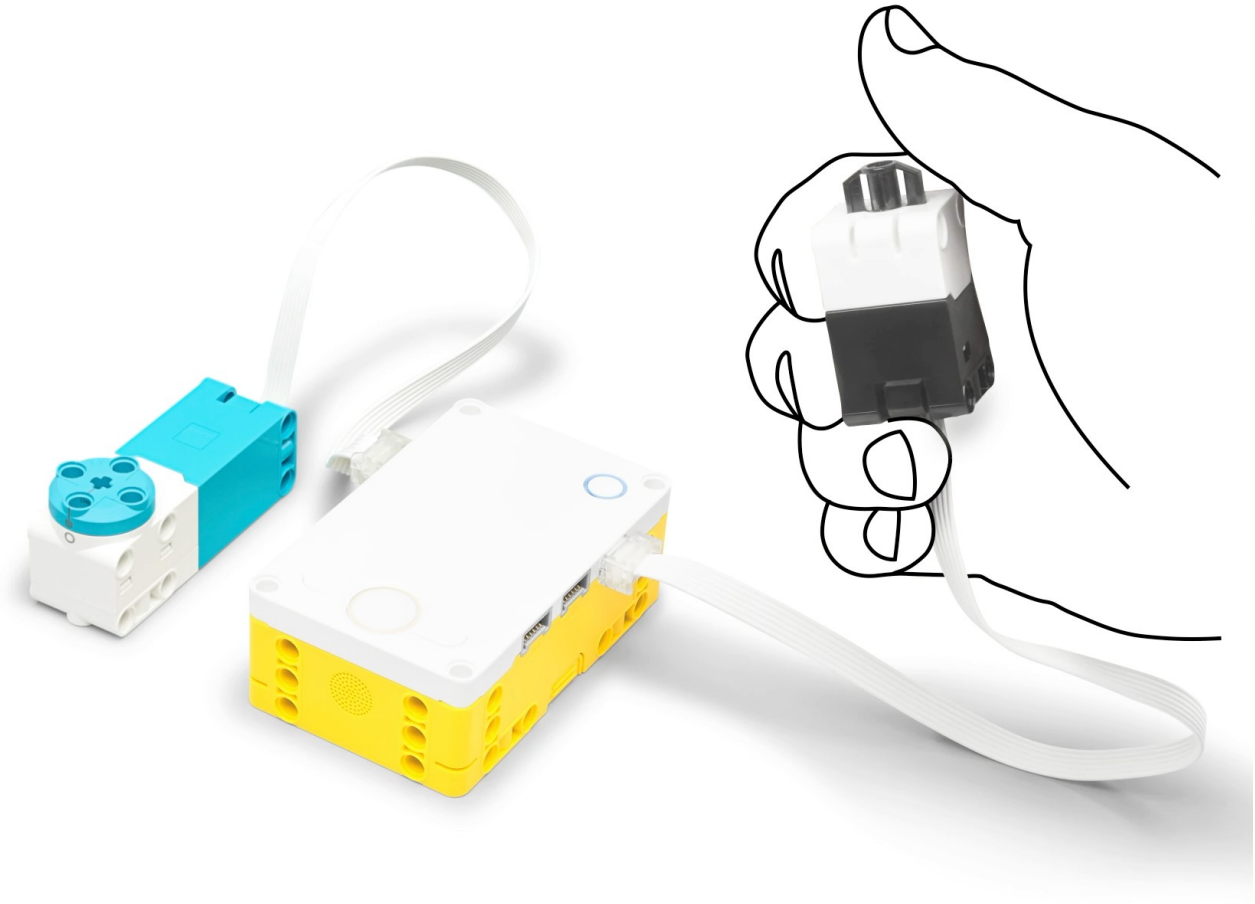
Défi

Arriverez-vous à modifier le code pour qu’il y ait différentes couleurs dans la liste `colors` ?

Contrôle du capteur

Dans les chapitres précédents, vous avez essayé d’utiliser des variables et des nombres aléatoires pour contrôler les moteurs et la lumière. Vous allez à présent utiliser une valeur de capteur pour contrôler un moteur.

Connectez un moteur au port A et un capteur de force au port B, puis essayez le programme ci-dessous.



```
import force_sensor
import motor
from hub import port

# Stockez la force du capteur de force dans une variable.
force = force_sensor.force(port.B)

# Inscrivez la variable dans la console.
print(force)

# Faites fonctionner le moteur et utilisez la variable pour régler la vitesse.
motor.run(port.A, force)
```

Appuyez sur le capteur de force pendant que le programme est en cours d'exécution. Cela n'a pas fait grand-chose, n'est-ce pas ? Heureusement, l'exemple utilise la fonction intégrée `print()` pour écrire la variable `force` dans la console, afin que vous puissiez facilement voir ce qui s'est mal passé.

La console

Parfois, votre programme ne fait pas ce que vous attendez de lui. Vous pouvez utiliser la fonction `print()` pour *déboguer* votre programme lorsque cela se produit. La fonction `print()` écrit tout ce que vous passez comme argument dans la fenêtre de la console sous l'éditeur de code, dans le cas présent la force exercée sur le capteur de force. Exécutez à nouveau le programme et notez la valeur qui apparaît dans la console.

Vous ne verrez qu'un seul nombre dans la console et, à moins que vous n'étiez en train d'appuyer sur le capteur de force lorsque vous avez démarré le programme, ce nombre sera `0`. Faire fonctionner un moteur à 0 degré par seconde ne fait pas grand-chose. Le problème vient donc du fait que le programme ne vérifie la valeur du capteur qu'une seule fois au début du programme. Pour mettre à jour la vitesse du moteur en fonction de la force, aussi longtemps que le programme s'exécute, vous devez réutiliser la boucle `while True`.

La console affiche également des messages d'erreur en cas de problème lors de l'exécution de votre programme. Une erreur courante se produit lorsque vous exécutez un programme pour contrôler un moteur ou lire un capteur qui n'est pas connecté. Déconnectez le capteur de force et exécutez le même programme une dernière fois. Vous verrez un message d'erreur dans la console vous informant qu'il y a eu un problème, quelle en était la nature et la ligne de code sur laquelle il s'est produit.

Correction des bogues

La console vous a aidé à trouver deux bogues. Reconnectez le capteur de force au port B pour corriger le deuxième bogue. Exécutez ensuite le programme ci-dessous qui corrige le premier bogue en *intégrant* le code dans une boucle `while True`.

```
import force_sensor
import motor
from hub import port

while True:
    # Stockez la force du capteur de force dans une variable.
    force = force_sensor.force(port.B)

    # Inscrivez la variable dans la console.
    print(force)

    # Faites fonctionner le moteur et utilisez la variable pour régler la
vitesse.
    motor.run(port.A, force)
```

Appuyez sur le capteur de force pendant que le programme est en cours d'exécution. Vous verrez le moteur accélérer ou ralentir en fonction de la force avec laquelle vous appuyez sur le capteur de force. Vous verrez également de nombreuses valeurs de variables écrites dans la console. La force exercée sur le capteur de force est mesurée en décinewtons (dN). La force maximale qu'il peut mesurer étant de 10 newtons, la valeur maximale en dN est de 100. Faire tourner un moteur à 100 degrés par seconde n'est toujours pas très rapide !

Valeurs de renvoi de fonction

Au lieu de stocker la valeur du capteur de force dans une variable, vous pouvez également définir une fonction qui *renvoie* cette valeur. Séparer ainsi les différentes parties de votre programme facilite l'organisation de votre code et la correction des bogues s'ils se produisent.

Le programme suivant définit une fonction `motor_velocity()` qui renvoie la vitesse souhaitée du moteur en fonction de la force exercée sur le capteur de force au lieu d'utiliser une variable.

```
import force_sensor
import motor
from hub import port
```

```
# Cette fonction renvoie la vitesse du moteur souhaitée.
def motor_velocity():
    # La vitesse est cinq fois supérieure à la force du capteur de force.
    return force_sensor.force(port.B) * 5

while True:
    # Faites fonctionner le moteur comme précédemment.
    # Utilisez la valeur de retour de la fonction `motor_velocity()` pour la
vitesse.
    motor.run(port.A, motor_velocity())
```

Appuyez sur le capteur de force pendant que le programme est en cours d'exécution. Vous verrez le moteur accélérer ou ralentir en fonction de la force avec laquelle vous appuyez sur le capteur de force. La fonction `motor_velocity()` multiplie la valeur de la force par 5, de sorte que la vitesse sera comprise entre 0 et 500 degrés par seconde.

Défi

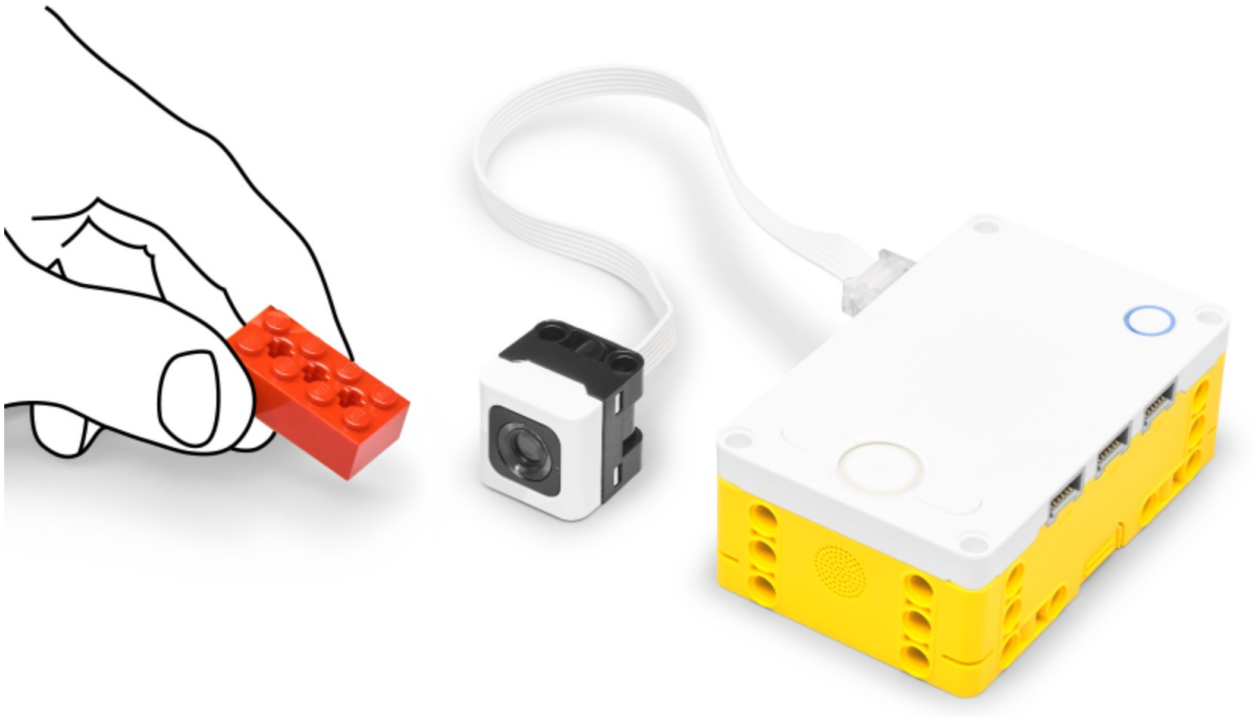
Arriverez-vous à modifier le code pour faire tourner le moteur à 1000 degrés par seconde lorsque le capteur de force est complètement enfoncé ?

Conditions du capteur

Vous avez utilisé la valeur du capteur pour directement contrôler un moteur, mais il est également possible de modifier le *flux* du programme à l'aide des *conditions* du capteur et d'une instruction `if`. L'instruction `if` est un élément essentiel de la programmation et le moyen le plus simple de contrôler le flux de votre programme.

Vous créez une instruction `if` en écrivant `if` suivi d'une expression logique et de deux-points. Les expressions logiques sont essentiellement des questions fermées, telles que « La couleur est-elle rouge ? » ou « Le bouton est-il enfoncé ? ». Si la réponse à la question est « oui », l'expression est évaluée comme étant `True`. Sinon, elle est `False`. Ce sont les deux valeurs *booléennes* utilisées dans Python, qui sont de type `bool`. Toutes les lignes de code avec le même niveau de mise en retrait après l'instruction `if` font partie du bloc de code qui s'exécute si l'expression est `True`.

Pour voir un exemple, connectez un capteur de couleur au port A et exécutez le programme ci-dessous.



```
from hub import port, sound
import color
import color_sensor
import runloop

async def main():
    while True:
        # Vérifiez si la couleur rouge est détectée.
        if color_sensor.color(port.A) == color.RED:
            # Si du rouge est détecté, un très long bip est émis.
            sound.beep(440, 1000000, 100)
            # Mettez le programme en pause lorsque le rouge est détecté.
            while color_sensor.color(port.A) == color.RED:
                await runloop.sleep_ms(1)
            # Arrêtez le son lorsque le rouge n'est plus détecté.
            sound.stop()

runloop.run(main())
```

Agitez une brique LEGO® rouge devant le capteur de couleur pendant que le programme est en cours d'exécution. Lorsque la couleur rouge est détectée, vous entendez un bip qui s'arrête lorsqu'elle ne l'est plus. L'exemple utilise une instruction `if` pour vérifier si la couleur détectée par le capteur de couleur est le rouge. Pour ce faire, il utilise l'*opérateur d'égalité* `==` avec la valeur de couleur du capteur de couleur à gauche et la constante `color.RED` à droite. (Notez qu'il y a deux signes `=` contrairement au signe unique `=` que vous avez utilisé pour attribuer des valeurs aux variables.) Si la valeur de couleur du capteur de couleur est identique à la constante `color.RED`, la condition est `True` et le bloc de code après la commande `if` s'exécute.

Il est important d'intégrer le code dans une boucle `while True`. Sinon, le capteur de couleur ne vérifiera la couleur que pendant une fraction de seconde au démarrage du programme. Jusqu'à présent, vous utilisiez la boucle `while` avec la constante `True` pour répéter le code à l'infini. Vous pouvez également utiliser la boucle `while` avec une expression logique pour répéter le code uniquement tant que cette expression a la valeur `True`. L'exemple ci-dessus utilise la même condition que l'instruction `if` dans la boucle interne `while` pour continuer à émettre le bip tant que la couleur rouge est détectée. Lorsque la condition n'est plus `True`, le Hub *quitte* la boucle `while` et exécute la ligne de code suivante pour arrêter le son.

À l'intérieur de la boucle interne `while`, remarquez la fonction `sleep_ms()` du module `runloop`. Cette fonction met en pause la coroutine `main()` pendant un certain nombre de millisecondes de manière *non bloquante*. Comme elle utilise le mot-clé `await`, d'autres tâches peuvent s'exécuter pendant que cette coroutine est en pause. Dans l'exemple, la pause est d'une milliseconde. Cela peut sembler très court, mais c'est suffisant pour que le Hub exécute de nombreuses coroutines simultanément. La fonction `sleep_ms()` du module `time`, que vous avez utilisée dans les chapitres précédents, met en pause le programme de manière *bloquante*. Cela signifie qu'elle met en pause l'ensemble du programme et pas seulement le bloc de code où vous l'appelez.

Quoi d'autre ?

Vous pouvez ajouter plus d'une condition en prolongeant l'instruction `if` avec une instruction `elif` qui vérifie une autre condition. Vous pouvez en ajouter autant que nécessaire. Elles suivent la même syntaxe que l'instruction `if`. La condition `elif` doit être au même niveau de mise en retrait que la première instruction `if`. Le mot-clé `elif` est suivi d'une expression logique et de deux-points. Mettez en retrait la ou les lignes de code suivantes qui doivent s'exécuter lorsque cette condition est `True`.

Parfois, aucune des conditions dans les instructions `if` et `elif` n'est `True`. Dans ce cas, vous pouvez exécuter du code en ajoutant une instruction `else` sans aucune condition. Cette opération s'exécute lorsque toutes les conditions précédentes sont `False`.

Par exemple, le programme ci-dessous ajoute des instructions `elif` et `else` pour qu'un bip soit également émis si vous appuyez sur le bouton gauche.

```
from hub import button, port, sound
import color
import color_sensor
import runloop

# Cette fonction renvoie `True` si le capteur de couleur détecte du rouge.
def red_detected():
    return color_sensor.color(port.A) == color.RED

# Cette fonction renvoie `True` si on appuie sur le bouton gauche.
def left_pressed():
    return button.pressed(button.LEFT) > 0

async def main():
    while True:
        if red_detected():
            # Si du rouge est détecté, un très long bip est émis.
```

```
            sound.beep(440, 1000000, 100)
            # Attendez que le rouge ne soit plus détecté.
            while red_detected():
                await runloop.sleep_ms(1)
        elif left_pressed():
            # Si on appuie sur le bouton gauche, un court bip est émis.
            sound.beep(880, 200, 100)
            # Attendez que le bouton gauche soit relâché.
            while left_pressed():
                await runloop.sleep_ms(1)
        else:
            # Sinon, arrêtez le son.
            sound.stop()

runloop.run(main())
```

Agitez une brique LEGO rouge devant le capteur de couleur et appuyez sur le bouton gauche du Hub pendant que le programme est en cours d'exécution. Vous entendrez un bip aussi longtemps que la couleur rouge est détectée, et un bip court chaque fois que vous appuyez sur le bouton gauche.

L'exemple définit deux fonctions pour effectuer les tests logiques et renvoyer le résultat. La fonction `red_detected()` vérifie si la couleur détectée par le capteur de couleur est rouge et renvoie le résultat `True` ou `False`. La fonction `left_pressed()` utilise la fonction `pressed()` du module `hub.button` et utilise l'opérateur `>` pour vérifier si la valeur est *supérieure à* `0` et renvoie le résultat.

Ici, le code dans l'instruction `if` est en grande partie le même que pour le premier exemple, mais il utilise maintenant la fonction `red_detected()` à deux endroits au lieu de répéter la condition pour les instructions `if` et `while`. L'instruction `elif` utilise la fonction `left_pressed()` pour vérifier si le bouton gauche du Hub est enfoncé pendant plus de `0` millisecondes.

Notez que l'instruction `elif` ne s'exécute que lorsque la condition de la première instruction `if` est `False`. Par conséquent, appuyer sur le bouton lorsque que le capteur de couleur détecte un objet rouge n'a aucun effet. Vous devez examiner attentivement l'ordre de vos conditions `if` et `elif`, et vérifier les plus importantes en premier. L'instruction `else` arrête le son si aucune des deux conditions n'est `True`.

Plusieurs conditions

Lorsque vous utilisez les instructions `if`/`elif`/`else` pour tester plusieurs conditions, un seul des blocs s'exécute. Ces conditions *s'excluent mutuellement*. Vous avez constaté que lorsque la couleur rouge était détectée, appuyer sur le bouton gauche n'avait aucun effet. Pour vraiment vérifier plusieurs conditions, vous devez le faire simultanément. Comme pour l'ajout de plusieurs blocs de mots, en Python, vous pouvez exécuter plusieurs coroutines avec la fonction `run()` du module `runloop`. Jusqu'à présent, vous l'appeliez avec la coroutine `main()` comme seul argument, mais il est possible de passer plusieurs coroutines en tant qu'arguments séparés par des virgules.

Par exemple, le programme ci-dessous divise le code qui vérifie la couleur détectée et le bouton enfoncé en deux coroutines. La fonction `run()` sur la dernière ligne de code démarre les deux coroutines en même temps.

```
from hub import button, port, sound
import color
import color_sensor
import runloop

# Cette fonction renvoie `True` si le capteur de couleur détecte du rouge.
def red_detected():
    return color_sensor.color(port.A) == color.RED

# Cette fonction renvoie `True` si on appuie sur le bouton gauche.
def left_pressed():
    return button.pressed(button.LEFT) > 0

# Cette coroutine vérifie en permanence si le capteur de couleur détecte du
rouge.
async def check_color():
    while True:
        # Attendez que le rouge soit détecté.
        while not red_detected():
            await runloop.sleep_ms(1)
        # Lorsqu'il est détecté, un très long bip est émis.
        sound.beep(440, 1000000, 100)
        # Attendez que le rouge ne soit plus détecté.
        while red_detected():
            await runloop.sleep_ms(1)
        # Lorsque le rouge n'est plus détecté, arrêtez le son.
        sound.stop()

# Cette coroutine vérifie en permanence si le bouton gauche est enfoncé.
async def check_button():
    while True:
        # Attendez jusqu'à ce qu'on appuie sur le bouton gauche.
        while not left_pressed():
            await runloop.sleep_ms(1)
        # Lorsqu'on appuie dessus, un court bip est émis.
        sound.beep(880, 200, 100)
        # Attendez que le bouton gauche soit relâché.
        while left_pressed():
            await runloop.sleep_ms(1)

# Exécutez les deux coroutines.
runloop.run(check_color(), check_button())
```

Agitez une brique LEGO rouge devant le capteur de couleur et appuyez sur le bouton gauche du Hub pendant que le programme est en cours d'exécution. Comme précédemment, vous entendrez un bip aussi longtemps que la couleur rouge est détectée, et un bip court chaque fois que vous appuyez sur le bouton gauche. Cette fois, il est possible d'appuyer sur le bouton gauche et d'entendre un bip pendant que la couleur rouge est détectée car les deux fonctions s'exécutent en même temps.

Lorsque vous créez vos propres coroutines, n'oubliez pas que :

- Vos coroutines doivent avoir au moins une commande `await`.
- Lorsque vous utilisez une boucle `while` *serrée*, utilisez `await runloop.sleep_ms(1)` à l'intérieur de la boucle pour donner aux autres coroutines la possibilité de démarrer et de s'exécuter.

Défi

Arriverez-vous à modifier le code pour détecter une autre couleur que le rouge ?

Étapes suivantes

Dans les chapitres précédents, vous

- avez appris les bases de l'utilisation de Python avec SPIKE Principal et comment utiliser la matrice lumineuse, la lumière, le haut-parleur et les boutons du Hub, ainsi que les moteurs, le capteur de couleur et le capteur de force.
- vous êtes familiarisé(e) avec les fonctions régulières et asynchrones, les variables locales et globales et les types de données tels que `int`, `bool`, `str`, `tuple` et `list`.
- avez utilisé les boucles `for` et `while`, ainsi que les instructions `if`/`elif`/`else` pour contrôler le flux de votre programme.
- avez appris à utiliser les commentaires dans votre code et, lorsque les choses tournent mal, à déboguer le programme.

C'est tout un exploit. Vous pouvez être fier(e) de vous !

Il existe des ressources supplémentaires auxquelles vous pouvez accéder pour en savoir plus sur l'utilisation de Python avec SPIKE Principal.

Guide de l'utilisateur Python

La section **Mise en route** a à peine abordé ce qu'il est possible de faire avec Python et SPIKE Principal. Explorez ces trois autres sections du guide de l'utilisateur Python.

1. **Exemples** Découvrez des exemples de programmes qui vous montrent comment utiliser Python pour résoudre diverses tâches. Copiez-les et essayez-les, puis modifiez-les en fonction de vos besoins.
2. **Modules de SPIKE Principal** Découvrez la documentation de toutes les fonctions et variables des modules SPIKE Principal comportant de courts exemples de leur utilisation.

Leçons sur Python

Sur le site Web LEGOeducation.com/lessons, sélectionnez le produit **SPIKE™ Principal avec Python**. Vous trouverez plusieurs plans de cours avec 6 à 8 leçons chacun (disponibles en anglais uniquement). Ces plus de 50 leçons couvrent un large éventail de sujets, du débogage au contrôle des capteurs, et de jeux simples jusqu'aux fonctions de données et mathématiques. Découvrez les nombreuses possibilités et devenez un(e) expert(e) en utilisant Python avec SPIKE Principal.

Défi

Créez un nouveau projet Python et commencez à coder !

Modules API

Application

Le module `app` est utilisé pour communiquer entre le hub et l'application

Sous-modules

Graphique à barres

Le module `bargraph` est utilisé pour créer des graphiques à barre dans l'application SPIKE

Pour utiliser le module, importez simplement le module `bargraph` comme suit:

```
from app import bargraph
```

Détails `bargraph`

Fonctions

change

change(color: int, value: float) -> None

Paramètres

---

color: int

Une couleur du module `color`

value: float

La valeur

clear_all

clear_all() -> None

Paramètres

---

get_value

get_value(color: int) -> Awaitable

Paramètres

---

color: int

Une couleur du module `color`

hide

hide() -> None

Paramètres

---

set_value

set_value(color: int, value: float) -> None

Paramètres

---

color: int

Une couleur du module `color`

value: float

La valeur

show

show(fullscreen: bool) -> None

Paramètres

---

fullscreen: bool
Mode Plein écran

Affichage

Le module `display` est utilisé pour afficher les images dans l'application SPIKE

Pour utiliser le module, importez simplement le module `display` comme suit:

```
from app import display
```

Détails `display`

Fonctions

hide
hide() -> None

Paramètres

---

image
image(image: int) -> None

Paramètres

---

image: int
L'identifiant de l'image à afficher. La plage d'images disponibles va de 1 à 21. Il y a des consts sur le module `display` pour ceci

show
show(fullscreen: bool) -> None

Paramètres

---

fullscreen: bool
Mode Plein écran

text
text(text: str) -> None

Paramètres

---

text: str
Le texte à afficher

Constantes

Constantes app.display

IMAGE_ROBOT_1 = 1

IMAGE_ROBOT_2 = 2

IMAGE_ROBOT_3 = 3

IMAGE_ROBOT_4 = 4

IMAGE_ROBOT_5 = 5

IMAGE_HUB_1 = 6

IMAGE_HUB_2 = 7

IMAGE_HUB_3 = 8

IMAGE_HUB_4 = 9

IMAGE_AMUSEMENT_PARK = 10

IMAGE_BEACH = 11

IMAGE_HAUNTED_HOUSE = 12

IMAGE_CARNIVAL = 13

IMAGE_BOOKSHELF = 14

IMAGE_PLAYGROUND = 15

IMAGE_MOON = 16

IMAGE_CAVE = 17

IMAGE_OCEAN = 18

IMAGE_POLAR_BEAR = 19

IMAGE_PARK = 20

IMAGE_RANDOM = 21

Graphique linéaire

Le module `linegraph` est utilisé pour faire des graphiques linéaires dans l'application SPIKE

Pour utiliser le module, importez simplement le module `linegraph` comme suit:

```
from app import linegraph
```

Détails `linegraph`

Fonctions

clear

clear(color: int) -> None

Paramètres

---

color: int
Une couleur du module `color`

clear_all
clear_all() -> None

Paramètres

---

get_average
get_average(color: int) -> Awaitable

Paramètres

---

color: int
Une couleur du module `color`

get_last
get_last(color: int) -> Awaitable

Paramètres

---

color: int
Une couleur du module `color`

get_max
get_max(color: int) -> Awaitable

Paramètres

---

color: int
Une couleur du module `color`

get_min
get_min(color: int) -> Awaitable

Paramètres

---

color: int
Une couleur du module `color`

hide
hide() -> None

Paramètres

---

plot
plot(color: int, x: float, y: float) -> None

Paramètres

---

color: int
Une couleur du module `color`

x: float
La valeur X

y: float
La valeur Y

show
show(fullscreen: bool) -> None

Paramètres

---

fullscreen: bool
Mode Plein écran

Musique

Le module `music` est utilisé pour faire de la musique dans l'application SPIKE

Pour utiliser le module, importez simplement le module `music` comme suit:

```
from app import music
```

Détails `music`

Fonctions

play_drum
play_drum(drum: int) -> None

Paramètres

---

drum: int
Le nom de la batterie. Consultez toutes les valeurs disponibles dans le module app.sound.

play_note
play_note(instrument: int, note: int, duration: int) -> None

Paramètres

---

instrument: int
Le nom de l'instrument. Consultez toutes les valeurs disponibles dans le module app.music.

note: int
La note midi à jouer (0-130)

duration: int
La durée en millisecondes

Constantes

Constantes app.music

DRUM_BASS = 2

DRUM_BONGO = 13

DRUM_CABASA = 15

DRUM_CLAVES = 9

DRUM_CLOSED_HI_HAT = 6

DRUM_CONGA = 14

DRUM_COWBELL = 11

DRUM_CRASH_CYMBAL = 4

DRUM_CUICA = 18

DRUM_GUIRO = 16

DRUM_HAND_CLAP = 8

DRUM_OPEN_HI_HAT = 5

DRUM_SIDE_STICK = 3

DRUM_SNARE = 1

DRUM_TAMBOURINE = 7

DRUM_TRIANGLE = 12

DRUM_VIBRASLAP = 17

DRUM_WOOD_BLOCK = 10

INSTRUMENT_BASS = 6

INSTRUMENT_BASSOON = 14

INSTRUMENT_CELLO = 8

INSTRUMENT_CHOIR = 15

INSTRUMENT_CLARINET = 10

INSTRUMENT_ELECTRIC_GUITAR = 5

INSTRUMENT_ELECTRIC_PIANO = 2

INSTRUMENT_FLUTE = 12

INSTRUMENT_GUITAR = 4

INSTRUMENT_MARIMBA = 19

INSTRUMENT_MUSIC_BOX = 17

INSTRUMENT_ORGAN = 3

INSTRUMENT_PIANO = 1

INSTRUMENT_PIZZICATO = 7

INSTRUMENT_SAXOPHONE = 11

INSTRUMENT_STEEL_DRUM = 18

INSTRUMENT_SYNTH_LEAD = 20

INSTRUMENT_SYNTH_PAD = 21

INSTRUMENT_TROMBONE = 9

INSTRUMENT_VIBRAPHONE = 16

INSTRUMENT_WOODEN_FLUTE = 13

Son

Le module `sound` est utilisé pour jouer des sons dans l'application SPIKE

Pour utiliser le module, importez simplement le module `sound` comme suit:

```
from app import sound
```

Détails `sound`

Fonctions

play
play(sound_name: str, volume: int = 100, pitch: int = 0, pan: int = 0) -> Awaitable

Jouer un son dans l'application SPIKE

Paramètres

---

sound_name: str
Le nom du son tel qu'il apparaît dans l'extension sonore Word Blocks

volume: int
Le volume (0-100)

pitch: int
La tonalité du son

pan: int
L'effet de panoramique détermine quel haut-parleur émet le son, « -100 » correspondant au haut-parleur de gauche uniquement, « 0 » à une diffusion équilibrée normale, et « 100 » au haut-parleur de droite uniquement.

set_attributes
set_attributes(volume: int, pitch: int, pan: int) -> None

Paramètres

---

volume: int
Le volume (0-100)

pitch: int
La tonalité du son

pan: int
L’effet de panoramique détermine quel haut-parleur émet le son, « -100 » correspondant au haut-parleur de gauche uniquement, « 0 » à une diffusion équilibrée normale, et « 100 » au haut-parleur de droite uniquement.

stop
stop() -> None

Paramètres

Couleur

Le module `color` contient toutes les constantes de couleur à utiliser avec les modules `color_matrix`, `color_sensor` et `light`.

Pour utiliser le module Couleur, ajoutez l’instruction d’importation suivante à votre projet :

```
import color
```

Constantes

Constantes color

BLACK = 0

MAGENTA = 1

PURPLE = 2

BLUE = 3

AZURE = 4

TURQUOISE = 5

GREEN = 6

YELLOW = 7

ORANGE = 8

RED = 9

WHITE = 10

UNKNOWN = -1

Matrice de couleurs

Pour utiliser le module Matrice de couleurs, ajoutez l’instruction d’importation suivante à votre projet :

```
import color_matrix
```

Toutes les fonctions du module doivent être appelées à l’intérieur du module `color_matrix` sous la forme d’un préfixe comme suit :

```
color_matrix.set_pixel(port.A, 1, 1, (color.BLUE, 10))
```

Fonctions

clear

clear(port: int) -> None

Désactiver tous les pixels d’une matrice de couleurs

```
from hub import port
import color_matrix

color_matrix.clear(port.A)
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

get_pixel

get_pixel(port: int, x: int, y: int) -> tuple[int, int]

Récupérer un pixel spécifique représenté par un tuple contenant la couleur et l’intensité

```
from hub import port
import color_matrix

# Imprimez la couleur et l’intensité du pixel 0,0 sur la matrice de couleur
connectée au port A
print(color_matrix.get_pixel(port.A, 0, 0))
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

x: int

La valeur X (0 - 2)

y: int

La valeur Y, plage (0 - 2)

set_pixel

set_pixel(port: int, x: int, y: int, pixel: tuple[color: int, intensity: int]) -> None

Modifier un seul pixel sur une matrice de couleurs

```
from hub import port
```

```
import color
import color_matrix

# Modifiez la couleur du pixel 0,0 sur la matrice de couleur connectée au port A
color_matrix.set_pixel(port.A, 0, 0, (color.RED, 10))

# Imprimez la couleur du pixel 0,0 sur la matrice de couleur connectée au port A
print(color_matrix.get_pixel(port.A, 0, 0)[0])
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

x: int

La valeur X (0 - 2)

y: int

La valeur Y, plage (0 - 2)

pixel: tuple[color: int, intensity: int]

Tuple contenant la couleur et l'intensité, c'est-à-dire la luminosité du pixel

show

show(port: int, pixels: list[tuple[int, int]]) -> None

Modifier tous les pixels à la fois sur une matrice de couleurs

```
from hub import port
import color
import color_matrix

# Mettez à jour tous les pixels sur la Matrice de couleur à l'aide de la
fonction Afficher

# Créez une liste avec 18 éléments (paires de couleur et intensité)
pixels = [(color.BLUE, 10)] * 9

# Mettez à jour tous les pixels pour afficher la même couleur et la même
intensité
color_matrix.show(port.A, pixels)
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

pixels: list[tuple[int, int]]

Une liste contenant les tuples de valeurs de couleur et d'intensité pour les 9 pixels.

Capteur de couleur

Le module `color_sensor` vous permet d'écrire du code qui réagit à des couleurs spécifiques ou à l'intensité de la lumière réfléchie.

Pour utiliser le module Capteur de couleurs, ajoutez l'instruction d'importation suivante à votre projet :

```
import color_sensor
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `color_sensor` sous la forme d'un préfixe comme suit :

```
color_sensor.reflection(port.A)
```

Le capteur de couleur peut reconnaître les couleurs suivantes :

Rouge
Vert
Bleu
Magenta
Jaune
Orange
Bleu ciel
Noir
Blanc

Fonctions

color

color(port: int) -> int

Renvoie la valeur de couleur de la couleur détectée. Utilisez le module `color` pour mapper la valeur de couleur sur une couleur spécifique.

```
import color_sensor
from hub import port
import color

if color_sensor.color(port.A) is color.RED:
    print("Red detected")
```

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

reflection

reflection(port: int) -> int

Récupère l'intensité de la lumière réfléchie (0-100 %).

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

rgbi

rgbi(port: int) -> tuple[int, int, int, int]

Récupère l'intensité globale des couleurs et l'intensité du rouge, du vert et du bleu.

Renvoie tuple[red: int, green: int, blue: int, intensity: int]

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

Appareil

Le module `device` vous permet d'écrire du code pour obtenir des informations sur les appareils branchés sur le hub.

Pour utiliser le module Appareil, ajoutez l'instruction d'importation suivante à votre projet :

```
import device
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `device` sous la forme d'un préfixe comme suit :

```
device.device_id(port.A)
```

Fonctions

data

data(port: int) -> tuple[int]

Récupérez les données LPF-2 brutes d'un appareil.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

device_id

device_id(port: int) -> int

Récupérez l'identifiant d'appareil d'un appareil. Chaque appareil a un identifiant basé sur son type.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

get_duty_cycle

get_duty_cycle(port: int) -> int

Récupérez le cycle d'utilisation d'un appareil. Les valeurs renvoyées sont comprises entre 0 et 10000

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

ready

ready(port: int) -> bool

Lorsqu'un appareil est connecté au hub, il peut s'écouler quelques instants avant qu'il ne soit prêt à accepter les demandes.
Utilisez `ready` pour tester l'état de préparation des appareils connectés.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

set_duty_cycle

set_duty_cycle(port: int, duty_cycle: int) -> None

Définissez le cycle d'utilisation d'un appareil. Plage 0 à 10000

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

duty_cycle: int
La valeur PWM (0-10000)

Capteur de distance

Le module `distance_sensor` vous permet d'écrire du code qui réagit à des distances spécifiques ou d'allumer le capteur de distance de différentes manières.

Pour utiliser le module Capteur de distance, ajoutez l'instruction d'importation suivante à votre projet :

```
import distance_sensor
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `distance_sensor` sous la forme d'un préfixe comme suit :

```
distance_sensor.distance(port.A)
```

Fonctions

clear

clear(port: int) -> None

Éteint tous les voyants du capteur de distance connecté à `port`.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

distance

distance(port: int) -> int

Récupérez la distance en millimètres capturée par le capteur de distance connecté à `port`. Si le capteur de distance ne peut pas lire une distance valide, il retournera -1.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

get_pixel

get_pixel(port: int, x: int, y: int) -> int

Récupérez l'intensité d'une lumière spécifique sur le capteur de distance connecté à `port`.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

x: int
La valeur X (0 - 3)

y: int
La valeur Y, plage (0 - 3)

set_pixel

set_pixel(port: int, x: int, y: int, intensity: int) -> None

Modifie l’intensité d’un voyant spécifique sur le capteur de distance connecté à `port`.

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

x: int
La valeur X (0 - 3)

y: int
La valeur Y, plage (0 - 3)

intensity: int
Luminosité d’éclairage du pixel

show

show(port: int, pixels: list[int]) -> None

Changez tous les voyants en même temps.

```
from hub import port
import distance_sensor

# Mettez à jour tous les voyants du capteur de distance à l’aide de la fonction Afficher

# Créez une liste avec 4 valeurs d’intensité identiques
pixels = [100] * 4

# Mettez à jour tous les pixels pour afficher la même intensité
distance_sensor.show(port.A, pixels)
```

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

pixels: bytes
Une liste contenant des valeurs d’intensité pour les 4 pixels.

Capteur de force

Le module `force_sensor` contient toutes les fonctions et constantes pour utiliser le capteur de force.

Pour utiliser le module Exécuter la boucle, ajoutez l’instruction d’importation suivante à votre projet :

```
import force_sensor
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `force_sensor` sous la forme d'un préfixe comme suit :

```
force_sensor.force(port.A)
```

Fonctions

force

force(port: int) -> int

Récupère la force mesurée en décinewton. Les valeurs sont comprises entre 0 et 100

```
from hub import port
import force_sensor


print(force_sensor.force(port.A))
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

pressed

pressed(port: int) -> bool

Teste si une pression a été exercée sur le bouton du capteur. Renvoie true si une pression a été exercée sur le capteur de force connecté au port.

```
from hub import port
import force_sensor


print(force_sensor.pressed(port.A))
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

raw

raw(port: int) -> int

Renvoie la valeur de force brute non calibrée du capteur de force connecté au port `port`

```
from hub import port
import force_sensor


print(force_sensor.raw(port.A))
```

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

Hub

Sous-modules

Bouton

Pour utiliser le module Bouton, ajoutez l’instruction d’importation suivante à votre projet :

```
from hub import button
```

Toutes les fonctions du module doivent être appelées à l’intérieur du module `button` sous la forme d’un préfixe comme suit :

```
button.pressed(button.LEFT)
```

Fonctions

pressed
int pressed(button: int) -> int

Ce module vous permet de réagir à une pression exercée sur les boutons du hub. Vous devez d’abord importer le module `button` pour utiliser les boutons.

```
from hub import button

left_button_press_duration = 0

# Attendez d’avoir appuyé sur le bouton gauche
while not button.pressed(button.LEFT):
    pass

# Tant que vous appuyez sur le bouton gauche, mettez à jour la variable
`left_button_press_duration`
while button.pressed(button.LEFT):
    left_button_press_duration = button.pressed(button.LEFT)

print("Left button was pressed for " + str(left_button_press_duration) + "
milliseconds")
```

Paramètres

---

button: int
Un bouton du sous-module `button` dans le module `hub`

Constantes

Constantes hub.button
LEFT = 1
Bouton gauche à côté du bouton d’alimentation du Hub SPIKE Principal

RIGHT = 2
Bouton droit à côté du bouton d'alimentation du Hub SPIKE Principal

Lumière

Le module `light` comprend des fonctions permettant de changer la couleur du voyant sur le Hub SPIKE Principal.

Pour utiliser le module Lumière, ajoutez l'instruction d'importation suivante à votre projet :

```
from hub import light
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `light` sous la forme d'un préfixe comme suit :

```
light.color(color.RED)
```

Fonctions

color

color(light: int, color: int) -> None

Modifiez la couleur d'un voyant sur le hub.

```
from hub import light
import color

# Changez le voyant en rouge
light.color(light.POWER, color.RED)
```

Paramètres

---

light: int

La lumière sur le hub

color: int

Une couleur du module `color`

Constantes

Constantes hub.light

POWER = 0
Bouton de mise sous tension. Sur SPIKE Principal, c'est le bouton situé entre les boutons gauche et droit.
CONNECT = 1
Voyant autour du bouton de connexion Bluetooth sur SPIKE Principal.

Matrice lumineuse

Pour utiliser le module Matrice lumineuse, ajoutez l'instruction d'importation suivante à votre projet :

```
from hub import light_matrix
```

Toutes les fonctions du module doivent être appelées à l’intérieur du module `light_matrix` sous la forme d’un préfixe comme suit :

```
light_matrix.write("Hello World")
```

Fonctions

clear

clear() -> None

Éteint tous les pixels de la matrice lumineuse.

```
from hub import light_matrix
import time
# Mettez à jour les pixels pour afficher une image sur la matrice lumineuse,
puis les désactiver à l’aide de la fonction Effacer

# Affichez un petit cœur
light_matrix.show_image(2)

# Attendez deux secondes
time.sleep_ms(2000)

# Éteignez le cœur
light_matrix.clear()
```

Paramètres

---

get_orientation

get_orientation() -> int

Récupérez l’orientation actuelle de la matrice lumineuse.
Peut être utilisé avec les constantes suivantes : `orientation.UP`, `orientation.LEFT`, `orientation.RIGHT`, `orientation.DOWN`

Paramètres

---

get_pixel

get_pixel(x: int, y: int) -> int

Récupérez l’intensité d’un pixel spécifique sur la matrice lumineuse.

```
from hub import light_matrix

# Affichez un cœur
light_matrix.show_image(1)

# Imprimez la valeur de l’intensité du pixel central
print(light_matrix.get_pixel(2, 2))
```

Paramètres

---

x: int

La valeur X, plage (0 - 4)

y: int

La valeur Y, plage (0 - 4)

set_orientation

set_orientation(top: int) -> int

Changez l’orientation de la matrice lumineuse. Tous les appels suivants utiliseront la nouvelle orientation.
Peut être utilisé avec les constantes suivantes : `orientation.UP`, `orientation.LEFT`, `orientation.RIGHT`, `orientation.DOWN`

Paramètres

---

top: int

Le côté supérieur du hub

set_pixel

set_pixel(x: int, y: int, intensity: int) -> None

Règle la luminosité d’un pixel (une des 25 LED) de la matrice lumineuse.

```
from hub import light_matrix
# Activez le pixel au centre du hub
light_matrix.set_pixel(2, 2, 100)
```

Paramètres

---

x: int

La valeur X, plage (0 - 4)

y: int

La valeur Y, plage (0 - 4)

intensity: int

Luminosité d’éclairage du pixel

show

show(pixels: list[int]) -> None

Changez tous les voyants en même temps.

```
from hub import light_matrix
# Mettez à jour tous les pixels sur la matrice lumineuse à l’aide de la fonction Afficher

# Créez une liste avec 25 valeurs d’intensité identiques
pixels = [100] * 25

# Mettez à jour tous les pixels pour afficher la même intensité
light_matrix.show(pixels)
```

Paramètres

---

pixels: Iterable

Une liste contenant des valeurs d’intensité lumineuse pour les 25 pixels.

show_image

show_image(image: int) -> None

Affichez l’une des images intégrées à l’écran.

```
from hub import light_matrix
# Mettez à jour les pixels pour afficher une image sur la matrice lumineuse à
l’aide de la fonction show_image

# Affichez un visage souriant
light_matrix.show_image(light_matrix.IMAGE_HAPPY)
```

Paramètres

---

image: int

L’identifiant de l’image à afficher. La plage d’images disponibles va de 1 à 67. Il y a des consts sur le module `light_matrix` pour ceci.

write

write(text: str, intensity: int = 100, time_per_character: int = 500) -> Awaitable

Affiche du texte sur la matrice lumineuse, une lettre à la fois, en défilant de droite à gauche, sauf s’il y a un seul caractère à afficher qui ne défilera pas

```
from hub import light_matrix
# Écrivez un message au hub
light_matrix.write("Hello, world!")
```

Paramètres

---

text: str

Le texte à afficher

intensity: int

Luminosité d’éclairage du pixel

time_per_character: int

Durée d’affichage de chaque caractère à l’écran

Constantes

Constantes hub.light_matrix

IMAGE_HEART = 1

IMAGE_HEART_SMALL = 2

IMAGE_HAPPY = 3

IMAGE_SMILE = 4

IMAGE_SAD = 5

IMAGE_CONFUSED = 6

IMAGE_ANGRY = 7

IMAGE_ASLEEP = 8

IMAGE_SURPRISED = 9

IMAGE_SILLY = 10

IMAGE_FABULOUS = 11

IMAGE_MEH = 12

IMAGE_YES = 13

IMAGE_NO = 14

IMAGE_CLOCK12 = 15

IMAGE_CLOCK1 = 16

IMAGE_CLOCK2 = 17

IMAGE_CLOCK3 = 18

IMAGE_CLOCK4 = 19

IMAGE_CLOCK5 = 20

IMAGE_CLOCK6 = 21

IMAGE_CLOCK7 = 22

IMAGE_CLOCK8 = 23

IMAGE_CLOCK9 = 24

IMAGE_CLOCK10 = 25

IMAGE_CLOCK11 = 26

IMAGE_ARROW_N = 27

IMAGE_ARROW_NE = 28

IMAGE_ARROW_E = 29

IMAGE_ARROW_SE = 30

IMAGE_ARROW_S = 31

IMAGE_ARROW_SW = 32

IMAGE_ARROW_W = 33

IMAGE_ARROW_NW = 34

IMAGE_GO_RIGHT = 35

IMAGE_GO_LEFT = 36

IMAGE_GO_UP = 37

IMAGE_GO_DOWN = 38

IMAGE_TRIANGLE = 39

IMAGE_TRIANGLE_LEFT = 40

IMAGE_CHESSBOARD = 41

IMAGE_DIAMOND = 42

IMAGE_DIAMOND_SMALL = 43

IMAGE_SQUARE = 44

IMAGE_SQUARE_SMALL = 45

IMAGE_RABBIT = 46

IMAGE_COW = 47

IMAGE_MUSIC_CROTCHET = 48

IMAGE_MUSIC_QUAVER = 49

IMAGE_MUSIC_QUAVERS = 50

IMAGE_PITCHFORK = 51

IMAGE_XMAS = 52

IMAGE_PACMAN = 53

IMAGE_TARGET = 54

IMAGE_TSHIRT = 55

IMAGE_ROLLERSKATE = 56

IMAGE_DUCK = 57

IMAGE_HOUSE = 58

IMAGE_TORTOISE = 59

IMAGE_BUTTERFLY = 60

IMAGE_STICKFIGURE = 61

IMAGE_GHOST = 62

IMAGE_SWORD = 63

IMAGE_GIRAFFE = 64

IMAGE_SKULL = 65

IMAGE_UMBRELLA = 66

IMAGE_SNAKE = 67

Détecteur de mouvement

Pour utiliser le module Détecteur de mouvement, ajoutez l'instruction d'importation suivante à votre projet :

```
from hub import motion_sensor
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `motion_sensor` sous la forme d'un préfixe comme suit :

```
motion_sensor.up_face()
```

Fonctions

acceleration

acceleration(raw_unfiltered: bool) -> tuple[int, int, int]

Renvoie un tuple contenant les valeurs d'accélération x, y et z sous forme d'entiers. Les valeurs sont en milli G, soit 1/1000 G

Paramètres

---

raw_unfiltered: bool

Si nous voulons que les données soient renvoyées brutes et non filtrées

angular_velocity

angular_velocity(raw_unfiltered: bool) -> tuple[int, int, int]

Renvoie un tuple contenant les valeurs de vitesse angulaire x, y et z sous forme d'entiers. Les valeurs sont des décidegrés par seconde

Paramètres

---

raw_unfiltered: bool

Si nous voulons que les données soient renvoyées brutes et non filtrées

gesture

gesture() -> int

Renvoie le mouvement reconnu.

Les valeurs possibles sont les suivantes :

```
motion_sensor.TAPPED
motion_sensor.DOUBLE_TAPPED
motion_sensor.SHAKEN
motion_sensor.FALLING
motion_sensor.UNKNOWN
```

Paramètres

---

get_yaw_face

get_yaw_face() -> int

Récupérez la face du Hub auquel le lacet est relatif.
Si vous placez le Hub sur une surface plate avec la face tournée vers le haut, seul le lacet se mettra à jour lorsque vous faites pivoter le Hub
`motion_sensor.TOP` Face du Hub SPIKE Principal comportant le port de chargement USB.
`motion_sensor.FRONT` Face du Hub SPIKE Principal comportant la matrice lumineuse.
`motion_sensor.RIGHT` Côté droit du Hub SPIKE Principal, en regardant la face avant du Hub.
`motion_sensor.BOTTOM` Côté du Hub SPIKE Principal où se trouve la batterie.
`motion_sensor.BACK` Face du Hub SPIKE Principal où se trouve le haut-parleur.
`motion_sensor.LEFT` Côté gauche du Hub SPIKE Principal, en regardant la face avant du Hub.

Paramètres

---

quaternion
quaternion() -> tuple[float, float, float, float]

Renvoie le quaternion d'orientation du Hub sous la forme tuple[w: float, x: float, y: float, z: float].

Paramètres

---

reset_tap_count
reset_tap_count() -> None

Réinitialisez le nombre de pressions renvoyé par la fonction `tap_count`

Paramètres

---

reset_yaw
reset_yaw(angle: int) -> None

Modifiez le décalage de l'angle de lacet.
L'angle défini sera la nouvelle valeur de lacet.

Paramètres

---

angle: int

set_yaw_face
set_yaw_face(up: int) -> bool

Modifiez la face de Hub utilisée comme face de lacet. Si vous placez le Hub sur une surface plate avec la face tournée vers le haut, seul le lacet se mettra à jour lorsque vous faites pivoter le Hub

Paramètres

---

up: int
Face du hub qui doit être définie comme face du hub orientée vers le haut.
Les valeurs disponibles sont les suivantes :

`motion_sensor.TOP` Face du Hub SPIKE Principal comportant le port de chargement USB.
`motion_sensor.FRONT` Face du Hub SPIKE Principal comportant la matrice lumineuse.
`motion_sensor.RIGHT` Côté droit du Hub SPIKE Principal, en regardant la face avant du hub.
`motion_sensor.BOTTOM` Côté du Hub SPIKE Principal où se trouve la batterie.
`motion_sensor.BACK` Face du Hub SPIKE Principal où se trouve le haut-parleur.
`motion_sensor.LEFT` Côté gauche du Hub SPIKE Principal, en regardant la face avant du hub.

stable
stable() -> bool

Que le Hub repose à plat ou non.

Paramètres

---

tap_count
tap_count() -> int

Renvoie le nombre de pressions reconnues depuis le démarrage du programme ou le dernier appel de `motion_sensor.reset_tap_count()`.

Paramètres

---

tilt_angles
tilt_angles() -> tuple[int, int, int]

Renvoie un tuple contenant des valeurs de tangage de lacet et de roulis sous forme d'entiers. Les valeurs sont en décidegrés

Paramètres

---

up_face
up_face() -> int

Renvoie la face du hub actuellement orientée vers le haut
`motion_sensor.TOP` Face du Hub SPIKE Principal comportant le port de chargement USB.
`motion_sensor.FRONT` Face du Hub SPIKE Principal comportant la matrice lumineuse.
`motion_sensor.RIGHT` Côté droit du Hub SPIKE Principal, en regardant la face avant du Hub.
`motion_sensor.BOTTOM` Côté du Hub SPIKE Principal où se trouve la batterie.
`motion_sensor.BACK` Face du Hub SPIKE Principal où se trouve le haut-parleur.
`motion_sensor.LEFT` Côté gauche du Hub SPIKE Principal, en regardant la face avant du Hub.

Paramètres

Constantes

Constantes hub.motion_sensor

TAPPED = 0

DOUBLE_TAPPED = 1

SHAKEN = 2

FALLING = 3

UNKNOWN = -1

TOP = 0
Face du Hub SPIKE Principal comportant la matrice lumineuse.
FRONT = 1
Face du Hub SPIKE Principal où se trouve le haut-parleur.
RIGHT = 2
Côté droit du Hub SPIKE Principal, en regardant la face avant du hub.
BOTTOM = 3
Côté du Hub SPIKE Principal où se trouve la batterie.
BACK = 4
Face du Hub SPIKE Principal comportant le port de chargement USB.
LEFT = 5
Côté gauche du Hub SPIKE Principal, en regardant la face avant du hub.

Port

Ce module contient des constantes qui permettent d'accéder facilement aux ports du Hub SPIKE Principal. Utilisez les constantes dans toutes les fonctions qui utilisent un paramètre `port`.

Pour utiliser le module Port, ajoutez l'instruction d'importation suivante à votre projet :

```
from hub import port
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `port` sous la forme d'un préfixe comme suit :

```
port.A
```

Constantes

---

hub.port Constantes

A = 0
Le port étiqueté « A » sur le Hub.
B = 1
Le port étiqueté « B » sur le Hub.
C = 2
Le port étiqueté « C » sur le Hub.
D = 3
Le port étiqueté « D » sur le Hub.
E = 4
Le port étiqueté « E » sur le Hub.

F = 5
Le port étiqueté « F » sur le Hub.

Haut-parleur

Pour utiliser le module Haut-parleur, ajoutez l'instruction d'importation suivante à votre projet :

```
from hub import sound
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `sound` sous la forme d'un préfixe comme suit :

```
sound.stop()
```

Fonctions

beep
beep(frequency: int = 440, duration: int = 500, volume: int = 100, *, attack: int = 0, decay: int = 0, sustain: int = 100, release: int = 0, transition: int = 10, waveform: int = WAVEFORM_SINE, channel: int = DEFAULT) -> Awaitable

Émet un bip sonore à partir du hub

Paramètres

---

frequency: int
La fréquence d'exécution

duration: int
La durée en millisecondes

volume: int
Le volume (0-100)

Arguments de mots-clés facultatifs :

attack: int
Temps nécessaire pour monter le niveau initial de zéro à la valeur de pic, à partir du moment où la touche est enfoncée.

decay: int
Temps nécessaire pour la descente suivante du niveau d'attaque au niveau de maintien désigné.

sustain: int
Le niveau pendant la séquence principale de la durée du son, jusqu'à ce que la touche soit relâchée.

release: int
Le temps nécessaire pour que le niveau se désintègre du niveau de maintien à zéro après la libération de la touche

transition: int
temps en millisecondes pour passer au son si quelque chose est déjà joué dans le canal

waveform: int
La forme d'onde synthétisée. Utilisez l'une des constantes du module `hub.sound`.

channel: int

Le canal souhaité pour jouer, les options sont `sound.DEFAULT` et `sound.ANY`

stop

stop() -> None

Arrête tout bruit provenant du hub

Paramètres

---

volume

volume(volume: int) -> None

Paramètres

---

volume: int

Le volume (0-100)

Constantes

Constantes hub.sound

ANY = -2

DEFAULT = -1

WAVEFORM_SINE = 1

WAVEFORM_SAWTOOTH = 3

WAVEFORM_SQUARE = 2

WAVEFORM_TRIANGLE = 1

Fonctions

device_uuid

device_uuid() -> str

Récupérez l’identifiant de l’appareil.

Paramètres

---

hardware_id

hardware_id() -> str

Récupérez l’identifiant du matériel.

Paramètres

---

power_off

power_off() -> int

Éteint le hub.

Paramètres

---

temperature

temperature() -> int

Récupérez la température du hub. Mesuré en décidegrés Celsius (d°C) qui est de 1/10 de degré Celsius (°C)

Paramètres

Moteur

Pour utiliser un moteur, ajoutez l'instruction d'importation suivante à votre projet :

```
import motor
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `motor` sous la forme d'un préfixe comme suit :

```
motor.run(port.A, 1000)
```

Fonctions

absolute_position

absolute_position(port: int) -> int

Obtenez la position absolue d'un moteur

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

get_duty_cycle

get_duty_cycle(port: int) -> int

Obtenez le pwm d'un moteur

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

relative_position

relative_position(port: int) -> int

Obtenez la position relative d’un moteur

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

reset_relative_position

reset_relative_position(port: int, position: int) -> None

Modifiez la position utilisée comme décalage lors de l’utilisation de la fonction `run_to_relative_position`.

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

position: int

Le degré du moteur

run

run(port: int, velocity: int, *, acceleration: int = 1000) -> None

Démarrez un moteur à vitesse constante

```
from hub import port
import motor, time

# Démarrez le moteur
motor.run(port.A, 1000)
```

Paramètres

---

port: int

Un port du sous-module `port` dans le module `hub`

velocity: int

La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

Arguments de mots-clés facultatifs :

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

run_for_degrees

run_for_degrees(port: int, degrees: int, velocity: int, *, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Faites tourner un moteur à un nombre spécifique de degrés
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes :

```
motor.READY
motor.RUNNING
motor.STALLED
motor.CANCELED
motor.ERROR
motor.DISCONNECTED
```

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

degrees: int
Le nombre de degrés

velocity: int
La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante

`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

run_for_time

run_for_time(port: int, duration: int, velocity: int, *, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Faites fonctionner un moteur pendant une durée limitée
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes :

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.ERROR`
`motor.DISCONNECTED`

```
from hub import port
import runloop
import motor

async def main():
    # Exécution à la vitesse 1000 pendant 1 seconde
    await motor.run_for_time(port.A, 1000, 1000)

    # Exécution à la vitesse 280 pendant 1 seconde
    await motor_pair.run_for_time(port.A, 1000, 280)

    # Exécution à la vitesse 280 pendant 10 secondes avec une décélération lente
    await motor_pair.run_for_time(port.A, 10000, 280, deceleration=10)

runloop.run(main())
```

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

duration: int
La durée en millisecondes

velocity: int
La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

run_to_absolute_position

run_to_absolute_position(port: int, position: int, velocity: int, *, direction: int, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Faites tourner un moteur sur une position en valeur absolue.
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes :

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.CANCELED`
`motor.ERROR`
`motor.DISCONNECTED`

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

position: int
Le degré du moteur

velocity: int
La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

Arguments de mots-clés facultatifs :

direction: int
La direction à prendre.
Les options sont les suivantes :

```
motor.CLOCKWISE
motor.COUNTERCLOCKWISE
motor.SHORTEST_PATH
motor.LONGEST_PATH
```

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

run_to_relative_position

run_to_relative_position(port: int, position: int, velocity: int, *, stop: int = BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Faites tourner un moteur dans une position par rapport à la position actuelle.
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes :

```
motor.READY
motor.RUNNING
```

`motor.STALLED`
`motor.CANCELED`
`motor.ERROR`
`motor.DISCONNECTED`

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

position: int
Le degré du moteur

velocity: int
La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

set_duty_cycle

set_duty_cycle(port: int, pwm: int) -> None

Démarrez un moteur avec un pwm spécifique

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

pwm: int
La valeur PWM (0-10000)

stop

stop(port: int, *, stop: int = BRAKE) -> None

Arrête un moteur

```
from hub import port
import motor, time

# Démarrez le moteur
motor.run(port.A, 1000)

# Attendez 2 secondes
time.sleep_ms(2000)

# Arrêtez le moteur
motor.stop(port.A)
```

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

velocity

velocity(port: int) -> int

Obtenez la vitesse (deg/sec) d'un moteur

Paramètres

---

port: int
Un port du sous-module `port` dans le module `hub`

Constantes

Constantes motor

READY = 1

RUNNING = 2

STALLED = -1

CANCELED = -2

ERROR = -3

DISCONNECTED = 0

COAST = 1

BRAKE = 2

HOLD = 3

CONTINUE = 0

SMART_COAST = 4

SMART_BRAKE = 5

CLOCKWISE = 0

COUNTERCLOCKWISE = 1

SHORTEST_PATH = 2

LONGEST_PATH = 3

Paire de moteurs

Le module `motor_pair` est utilisé pour faire fonctionner les moteurs de manière synchronisée. Ce mode est optimal pour créer des bases de transmission où vous voudriez qu'une paire de moteurs démarre et s'arrête en même temps.

Pour utiliser le module, importez simplement le module `motor_pair` comme suit:

```
import motor_pair
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `motor_pair` sous la forme d'un préfixe comme suit :

```
motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)
```

Fonctions

move

move(pair: int, steering: int, *, velocity: int = 360, acceleration: int = 1000) -> None

Déplacez une paire de moteurs à vitesse constante jusqu'à ce qu'une nouvelle commande soit donnée.

```
from hub import port
import runloop
import motor_pair

async def main():
    # Jumelez les moteurs sur les ports A et B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    await runloop.sleep_ms(2000)

    # Déplacement tout droit à la vitesse par défaut
    motor_pair.move(motor_pair.PAIR_1, 0)

    await runloop.sleep_ms(2000)

    # Déplacement tout droit à une vitesse spécifique
    motor_pair.move(motor_pair.PAIR_1, 0, velocity=280)

    await runloop.sleep_ms(2000)

    # Déplacement tout droit à une vitesse et une accélération spécifiques
    motor_pair.move(motor_pair.PAIR_1, 0, velocity=280, acceleration=100)

runloop.run(main())
```

Paramètres

---

pair: int

L'emplacement d'appairage de la paire de moteurs.

steering: int

La direction (-100 à 100)

Arguments de mots-clés facultatifs :

velocity: int

La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

acceleration: int

L'accélération (deg/sec^2) (1 - 10000)

move_for_degrees

move_for_degrees(pair: int, degrees: int, steering: int, *, velocity: int = 360, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Déplacez une paire de moteurs à vitesse constante pendant un nombre spécifique de degrés. Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes du module moteur :

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.CANCELED`
`motor.ERROR`
`motor.DISCONNECTED`

```
from hub import port
import runloop
import motor_pair

async def main():
    # Jumelez les moteurs sur les ports A et B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Déplacement tout droit à la vitesse par défaut pendant 90 degrés
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 90, 0)

    # Déplacement tout droit à une vitesse spécifique
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 360, 0, velocity=280)

    # Déplacement tout droit à une vitesse spécifique avec une décélération
lente
    await motor_pair.move_for_degrees(motor_pair.PAIR_1, 360, 0, velocity=280,
deceleration=10)

runloop.run(main())
```

Paramètres

---

pair: int

L'emplacement d'appairage de la paire de moteurs.

degrees: int

Le nombre de degrés

steering: int

La direction (-100 à 100)

Arguments de mots-clés facultatifs :

velocity: int

La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

move_for_time

move_for_time(pair: int, duration: int, steering: int, *, velocity: int = 360, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Déplacez une paire de moteurs à vitesse constante pendant une durée spécifique.
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes du module moteur :

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.CANCELED`
`motor.ERROR`
`motor.DISCONNECTED`

```
from hub import port
import runloop
import motor_pair

async def main():
    # Jumelez les moteurs sur les ports A et B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Déplacement tout droit à la vitesse par défaut pendant 1 seconde
    await motor_pair.move_for_time(motor_pair.PAIR_1, 1000, 0)

    # Déplacement tout droit à une vitesse spécifique pendant 1 seconde
    await motor_pair.move_for_time(motor_pair.PAIR_1, 1000, 0, velocity=280)
```

```
    # Déplacement tout droit à une vitesse spécifique pendant 10 secondes avec
une décélération lente
    await motor_pair.move_for_time(motor_pair.PAIR_1, 10000, 0, velocity=280,
deceleration=10)

runloop.run(main())
```

Paramètres

---

pair: int
L’emplacement d’appairage de la paire de moteurs.

duration: int
La durée en millisecondes

steering: int
La direction (-100 à 100)

Arguments de mots-clés facultatifs :

velocity: int
La vitesse en degrés/sec

Les plages de valeurs dépendent du type de moteur.

Petit moteur (Essentiel) : -660 à 660
Moteur moyen : -1110 à 1110
Gros moteur : -1050 à 1050

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu’à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l’arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu’à ce qu’il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu’à l’arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l’arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L’accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

move_tank

move_tank(pair: int, left_velocity: int, right_velocity: int, *, acceleration: int = 1000) -> None

Effectuez un déplacement du robot sur une paire de moteurs à vitesse constante jusqu'à ce qu'une nouvelle commande soit donnée.

```
from hub import port
import runloop
import motor_pair

async def main():
    # Jumelez les moteurs sur les ports A et B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Déplacement tout droit à la vitesse par défaut
    motor_pair.move_tank(motor_pair.PAIR_1, 1000, 1000)

    await runloop.sleep_ms(2000)

    # Tournez à droite
    motor_pair.move_tank(motor_pair.PAIR_1, 0, 1000)

    await runloop.sleep_ms(2000)

    # Effectuez un tour du robot
    motor_pair.move_tank(motor_pair.PAIR_1, 1000, -1000)

runloop.run(main())
```

Paramètres

---

pair: int
L'emplacement d'appairage de la paire de moteurs.

left_velocity: int
La vitesse (deg/sec) du moteur gauche.

right_velocity: int
La vitesse (deg/sec) du moteur droit.

Arguments de mots-clés facultatifs :

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

move_tank_for_degrees

move_tank_for_degrees(pair: int, degrees: int, left_velocity: int, right_velocity: int, *, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Effectuez un déplacement du robot sur une paire de moteurs à vitesse constante jusqu'à ce qu'une nouvelle commande soit donnée.
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes du module moteur :

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.CANCELED`
`motor.ERROR`
`motor.DISCONNECTED`

```
from hub import port
import runloop
import motor_pair

async def main():
    # Jumelez les moteurs sur les ports A et B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Déplacement tout droit à la vitesse par défaut pour 360 degrés
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 360, 1000, 1000)

    # Tournez à droite à 180 degrés
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 180, 0, 1000)

    # Effectuez un tour du robot à 720 degrés
    await motor_pair.move_tank_for_degrees(motor_pair.PAIR_1, 720, 1000, -1000)

runloop.run(main())
```

Paramètres

---

pair: int
L’emplacement d’appairage de la paire de moteurs.

degrees: int
Le nombre de degrés

left_velocity: int
La vitesse (deg/sec) du moteur gauche.

right_velocity: int
La vitesse (deg/sec) du moteur droit.

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu’à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l’arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu’à ce qu’il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu’à l’arrêt, puis rouler en roue libre et

compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

move_tank_for_time

move_tank_for_time(pair: int, left_velocity: int, right_velocity: int, duration: int, *, stop: int = motor.BRAKE, acceleration: int = 1000, deceleration: int = 1000) -> Awaitable

Effectuez un déplacement du robot sur une paire de moteurs à vitesse constante pendant une durée spécifique.
Lorsqu'il est attendu, renvoie un état du mouvement qui correspond à l'une des constantes suivantes du module moteur :

`motor.READY`
`motor.RUNNING`
`motor.STALLED`
`motor.CANCELLED`
`motor.ERROR`
`motor.DISCONNECTED`

```
from hub import port
import runloop
import motor_pair

async def main():
    # Jumelez les moteurs sur les ports A et B
    motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)

    # Déplacement tout droit à la vitesse par défaut pendant 1 seconde
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 1000, 1000, 1000)

    # Tournez à droite pendant 3 secondes
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 0, 1000, 3000)

    # Effectuez un tour du robot pendant 2 secondes
    await motor_pair.move_tank_for_time(motor_pair.PAIR_1, 1000, -1000, 2000)

runloop.run(main())
```

Paramètres

---

pair: int
L'emplacement d'appairage de la paire de moteurs.

duration: int
La durée en millisecondes

left_velocity: int
La vitesse (deg/sec) du moteur gauche.

right_velocity: int
La vitesse (deg/sec) du moteur droit.

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

acceleration: int
L'accélération (deg/sec²) (1 - 10000)

deceleration: int
La décélération (deg/sec²) (1 - 10000)

pair

pair(pair: int, left_motor: int, right_motor: int) -> None

Jumelez deux moteurs (`left_motor` & `right_motor`) et stockez les moteurs appariés dans `pair`.
Utilisez `pair` dans tous les appels de fonction associés ultérieurs `motor_pair`.

```
import motor_pair
from hub import port

motor_pair.pair(motor_pair.PAIR_1, port.A, port.B)
```

Paramètres

---

pair: int
L'emplacement d'appairage de la paire de moteurs.

left_motor: int
Le port du moteur gauche. Utilisez le sous-module `port` dans le module `hub`.

right_motor: int
Le port du moteur droit. Utilisez le sous-module `port` dans le module `hub`.

stop

stop(pair: int, *, stop: int = motor.BRAKE) -> None

Arrête une paire de moteurs.

```
import motor_pair

motor_pair.stop(motor_pair.PAIR_1)
```

Paramètres

---

pair: int
L'emplacement d'appairage de la paire de moteurs.

Arguments de mots-clés facultatifs :

stop: int
Le comportement du moteur après son arrêt. Utilisez les constantes du module `motor`.

Les valeurs possibles sont les suivantes :
`motor.COAST` pour un déplacement en roue libre (coast) du moteur jusqu'à un arrêt
`motor.BRAKE` pour freiner et continuer à freiner après l'arrêt
`motor.HOLD` pour indiquer au moteur de maintenir sa position
`motor.CONTINUE` pour indiquer au moteur de continuer à tourner à la vitesse à laquelle il tourne jusqu'à ce qu'il reçoive une autre commande
`motor.SMART_COAST` pour faire freiner le moteur jusqu'à l'arrêt, puis rouler en roue libre et compenser les imprécisions dans la commande suivante
`motor.SMART_BRAKE` pour faire freiner le moteur et continuer à freiner après l'arrêt et compenser les imprécisions dans la commande suivante

unpair

unpair(pair: int) -> None

Dissociez une paire de moteurs.

```
import motor_pair

motor_pair.unpair(motor_pair.PAIR_1)
```

Paramètres

---

pair: int
L'emplacement d'appairage de la paire de moteurs.

Constantes

Constantes motor_pair

PAIR_1 = 0
Première paire de moteurs
PAIR_2 = 1
Deuxième paire de moteurs
PAIR_3 = 2
Troisième paire de moteurs

Orientation

Le module `orientation` contient toutes les constantes d'orientation à utiliser avec le module `light_matrix`.

Pour utiliser le module d'orientation, ajoutez l'instruction d'importation suivante à votre projet :

```
import orientation
```

Constantes

Constantes orientation

UP = 0

RIGHT = 1

DOWN = 2

LEFT = 3

Exécuter la boucle

Le module `runloop` contient toutes les fonctions et constantes requises pour utiliser l'exécution de boucle.

Pour utiliser le module Exécuter la boucle, ajoutez l'instruction d'importation suivante à votre projet :

```
import runloop
```

Toutes les fonctions du module doivent être appelées à l'intérieur du module `runloop` sous la forme d'un préfixe comme suit :

```
runloop.run(some_async_function())
```

Fonctions

run

run(*functions: Awaitable) -> None

Démarrez un nombre quelconque de fonctions `async` parallèles. C'est la fonction que vous devez utiliser pour créer des programmes avec une structure similaire à Word Blocks.

Paramètres

***functions: awaitable**

The functions to run

sleep_ms

sleep_ms(duration: int) -> Awaitable

Suspendez l'exécution de l'application pendant un nombre quelconque de millisecondes.

```
from hub import light_matrix
import runloop

async def main():
    light_matrix.write("Hi!")
    # Attendez dix secondes
    await runloop.sleep_ms(10000)
    light_matrix.write("Are you still here?")

runloop.run(main())
```

Paramètres

---

duration: int

La durée en millisecondes

until

until(function: Callable[[], bool], timeout: int = 0) -> Awaitable

Renvoie un attendable qui sera émis lorsque la condition de la fonction ou du lambda passé est `True` ou lorsqu'elle expire

```
import color_sensor
import color
from hub import port
import runloop

def is_color_red():
    return color_sensor.color(port.A) is color.RED

async def main():
    # Attendez que le capteur de couleur voit le rouge
    await runloop.until(is_color_red)
    print("Red!")

runloop.run(main())
```

Paramètres

---

function: Callable[[], bool]

Un appelable sans paramètre qui renvoie `True` ou `False`.
Un appelable est tout ce qui peut être appelé, donc un `def` ou un `lambda`

timeout: int

Un délai d’expiration de la fonction en millisecondes.

Si l’appelable ne renvoie pas `True` dans le délai d’expiration, le `until` reste résolu après le délai d’expiration.

0 signifie qu’il n’y a pas de délai d’expiration, dans ce cas, il ne sera pas résolu tant que l’appelable n’a pas renvoyé `True`