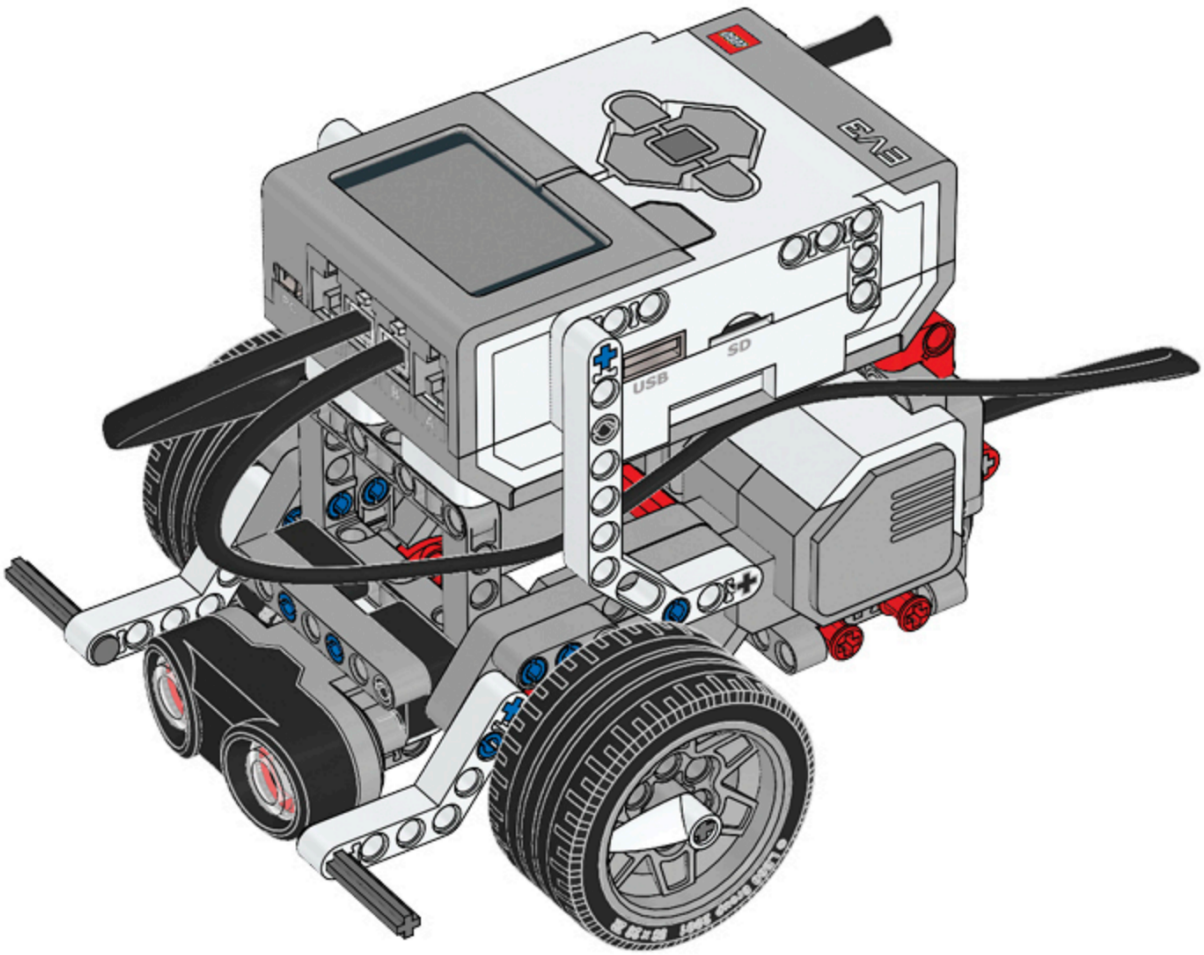
Cours de MicroPython

basé sur

Lego Education

Mindstorms EV3



©Hainaut Patrick 2026 - www.coursonline.be

TABLE DES MATIÈRES

PREMIÈRE PARTIE 4

COURS 1 : DÉCOUVERTE DE MICROPYTHON ET DU LEGO EV3 5

COURS 2 : LES VARIABLES ET LES TYPES EN MICROPYTHON AVEC LEGO EV3 37

COURS 3 : CALCULS, OPÉRATEURS ET DÉPLACEMENTS AVEC LEGO EV3 52

COURS 4 : LES CONDITIONS `IF`, `ELIF` ET `ELSE` AVEC LEGO EV3 69

COURS 5 : LES BOUCLES `WHILE` AVEC LEGO EV3 87

COURS 6 : LES BOUCLES `FOR` AVEC LEGO EV3 113

COURS 7 : LES MOTEURS EV3 AVEC MICROPYTHON 135

COURS 8 : CONTRÔLER DEUX MOTEURS ET DÉPLACER LE ROBOT EV3 156

COURS 9 : UTILISER `DRIVEBASE` POUR DÉPLACER PRÉCISÉMENT LE ROBOT EV3 178

COURS 10 : LES FONCTIONS AVEC MICROPYTHON ET LEGO EV3 201

COURS 11 : PARAMÈTRES, `RETURN` ET FONCTIONS RÉUTILISABLES AVEC LEGO EV3 227

COURS 12 : PROJET DE SYNTHÈSE AVANT LES CAPTEURS 254

COURS 13 : LE CAPTEUR TACTILE AVEC LEGO EV3 284

COURS 14 : LE CAPTEUR ULTRASON AVEC LEGO EV3 307

COURS 15 : LE CAPTEUR DE COULEUR AVEC LEGO EV3 337

COURS 16 : DÉTECTION DE LIGNE ET SUIVI DE LIGNE SIMPLE AVEC LEGO EV3 364

COURS 17 : LE SUIVI DE LIGNE PROPORTIONNEL AVEC LEGO EV3 388

COURS 18 : LISTES, MESURES ET HISTORIQUE DU ROBOT AVEC MICROPYTHON 416

COURS 19 : LE CONTRÔLEUR PID AVEC LEGO EV3 450

COURS 20 : LES MACHINES À ÉTATS AVEC LEGO EV3 483

COURS 21 : DÉBOGAGE ET DIAGNOSTIC D'UN ROBOT LEGO EV3 515

COURS 22 : COMBINER PLUSIEURS CAPTEURS ET GÉRER LES PRIORITÉS AVEC LEGO EV3 545

COURS 23 : LE GYROSCOPE ET LES ROTATIONS PRÉCISES AVEC LEGO EV3 576

COURS 24 : CONCEPTION DU PROJET FINAL AUTONOME DE LA PREMIÈRE PARTIE AVEC LEGO EV3 608

COURS 25 : PROJET FINAL DE LA PREMIÈRE PARTIE, ESSAIS ET COMPÉTITION AVEC LEGO EV3 645

DEUXIEME PARTIE 684

COURS 26 : COMPRENDRE LA CONNECTIVITE DE LA BRIQUE EV3 685

COURS 27 : ASSOCIER DEUX BRIQUES EV3 EN BLUETOOTH 700

COURS 28 : ENVOYER DES MESSAGES ENTRE DEUX BRIQUES EV3 715

COURS 29 : TRANSFORMER UNE BRIQUE EV3 EN TELECOMMANDE BLUETOOTH 738

COURS 30 : DECOUVRIR LES PERIPHERIQUES D'ENTREE LINUX 767

COURS 31 : CONNECTER UNE MANETTE BLUETOOTH A LA BRIQUE EV3 791

COURS 32 : LIRE LES BOUTONS D'UNE MANETTE BLUETOOTH 814

COURS 33 : UTILISER LES JOYSTICKS ANALOGIQUES D'UNE MANETTE 838

COURS 34 : CONSTRUIRE UN ROBOT EV3 RADIOCOMMANDE PAR MANETTE BLUETOOTH 866

COURS 35 : RESEAU, SSH ET PROJET FINAL DU ROBOT CONNECTE 894

CONCLUSION GENERALE 920

Première Partie

Cours 1 : Découverte de MicroPython et du LEGO EV3

Objectifs du cours

1. comprendre ce qu'est un robot
2. identifier les principaux éléments de la brique EV3
3. comprendre le rôle d'un programme
4. découvrir Python et MicroPython
5. créer un premier programme dans Visual Studio Code
6. envoyer un programme vers la brique EV3
7. faire produire un son à l'EV3
8. utiliser `wait()`
9. comprendre l'ordre d'exécution des instructions
10. modifier et tester un programme simple

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 1 ordinateur avec Visual Studio Code
3. MicroPython installé sur la brique EV3
4. l'environnement EV3 MicroPython configuré
5. un câble USB ou une connexion adaptée à la brique EV3

1. Introduction

Pendant cette formation, nous allons apprendre à programmer un robot LEGO Mindstorms EV3 avec Python.

Nous commencerons par des programmes extrêmement simples, puis nous ajouterons progressivement :

- `variables`
- `calculs`
- `conditions`
- `boucles`
- `fonctions`
- `moteurs`
- `capteurs`
- `suivi de ligne`
- `régulation`
- `robot autonome`

À la fin de la formation, nous serons capables de construire un robot qui observe son environnement et prend lui-même certaines décisions.

2. Qu’est-ce qu’un robot ?

Un robot est une machine capable d’effectuer des actions grâce à un programme.

Un robot peut généralement :

```
PERCEVOIR
   ↓
RÉFLÉCHIR
   ↓
AGIR
```

Pour cela, il possède trois grandes catégories d’éléments.

- PROGRAMME (CERVEAU)
- CAPTEURS
- ACTIONNEURS

3. Les capteurs

Les capteurs permettent au robot de recevoir des informations sur son environnement.

Par exemple :

- capteur tactile



- capteur ultrason



- capteur de couleur



- gyroscope



Un capteur peut répondre à des questions comme :

- Est-ce que quelque chose me touche ?
- À quelle distance se trouve l'obstacle ?
- Quelle couleur se trouve sous moi ?
- De combien de degrés ai-je tourné ?

4. Les actionneurs



Les actionneurs permettent au robot d'agir.

Sur l'EV3, les moteurs sont les principaux actionneurs.



Ils permettent par exemple de :

- faire avancer le robot
- tourner
- faire fonctionner un bras
- ouvrir une pince

La brique EV3 possède également un haut-parleur.

Nous allons justement l'utiliser dans ce cours.

5. Le programme

Entre les capteurs et les actionneurs se trouve le cerveau qui exécute le programme.

Le programme contient les instructions que le robot doit exécuter.

Par exemple :

- `Faire un son`
- `Attendre une seconde`
- `Faire un deuxième son`

Le robot exécute ces instructions dans l'ordre.

6. Le principe général

Nous pouvons représenter le robot ainsi :

```
CAPTEURS
  ↓
PROGRAMME (CERVEAU)
  ↓
MOTEURS
```

Ou plus généralement :

```
INFORMATION
  ↓
DÉCISION
  ↓
ACTION
```

Cette logique sera présente pendant toute la formation.

7. Découverte de la brique EV3

La brique EV3 est le cerveau du robot.

Elle permet notamment de :

- exécuter le programme
- recevoir les informations des capteurs
- commander les moteurs
- afficher des informations
- produire des sons



Voici ses caractéristiques :

- Batterie rechargeable (pour la version éducation 45544)
- Display de 128x128 pixels en monochrome
- 4 ports d'entrée pour capteurs
- 4 ports de sortie pour moteurs
- Navigation 6 boutons rétroéclairés via 3 couleurs de LED (vert, rouge, orange)
- Speaker
- Connectivité mini-USB et Bluetooth
- Port USB2 pour connecter une clé Wifi ou chaîner plusieurs EV3
- Emplacement Carte micro-SD
- Programmation en blocs et en MicroPython

8. Les ports moteurs

La brique EV3 possède quatre ports destinés principalement aux moteurs.

- A
- B
- C
- D



Nous les utiliserons plus tard avec des instructions comme :

```
Motor(Port.B)
```

9. Les ports capteurs

La brique possède également quatre ports destinés principalement aux capteurs.

- 1
- 2
- 3
- 4



Dans Python, nous les désignerons par exemple avec :

```
Port.S1
```

ou :

```
Port.S4
```

10. Exercice pratique 1

Observer une brique EV3 et identifier :

- les ports A B C D
- les ports 1 2 3 4
- l'écran
- les boutons
- le haut-parleur
- les connexions

L'objectif est de savoir retrouver rapidement les ports lors des prochains cours.

11. Qu'est-ce que Python ?

Python est un langage de programmation.

Un langage de programmation permet d'écrire des instructions compréhensibles par un ordinateur.

Par exemple :

```
print("Bonjour")
```

Cette instruction demande d'afficher :

```
Bonjour
```

Python est un langage de haut niveau qui est beaucoup plus léger que Java et qui permet le développement rapide d'applications.

C'est un langage interprété et il a besoin pour être exécuté d'un « moteur logiciel » qui va comprendre les instructions en Python.

On est actuellement à la version 3 de Python.

12. Qu'est-ce que MicroPython ?

MicroPython est une version de Python conçue pour fonctionner sur de petits systèmes électroniques.

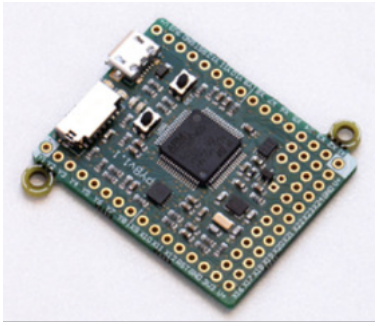
Dans notre cas, nous allons utiliser une version adaptée à l'EV3.

Nous pouvons donc programmer le robot avec une syntaxe proche de Python classique.

13. microcontrôleurs compatibles

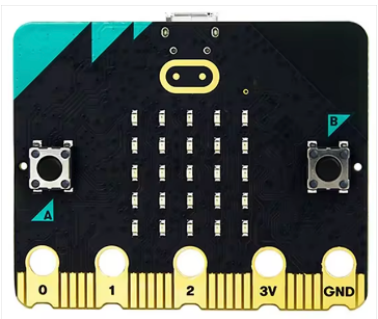
Parmi les microcontrôleurs pouvant se programmer sous MicroPython, nous trouvons :

- les cartes pyboard développées par le collectif MycroPython
- La plupart des cartes esp32 et stm32 sont compatibles
- La carte microbit
- les cartes Raspberry Pi Pico et Pico2
- Au niveau Arduino; la nano, la giga, la primo sont compatibles mais pas la uno ni la mega
- les briques Lego Mindstorms et Spike











14. Pourquoi apprendre Python ?

Python est utilisé dans de nombreux domaines.

Par exemple :

- automatisation
- administration système
- réseau
- analyse de données
- intelligence artificielle
- robotique
- développement

Le but n'est donc pas uniquement de programmer le LEGO EV3.

Les notions apprises pourront être réutilisées ailleurs.

15. Visual Studio Code

Nous allons écrire nos programmes dans Visual Studio Code.

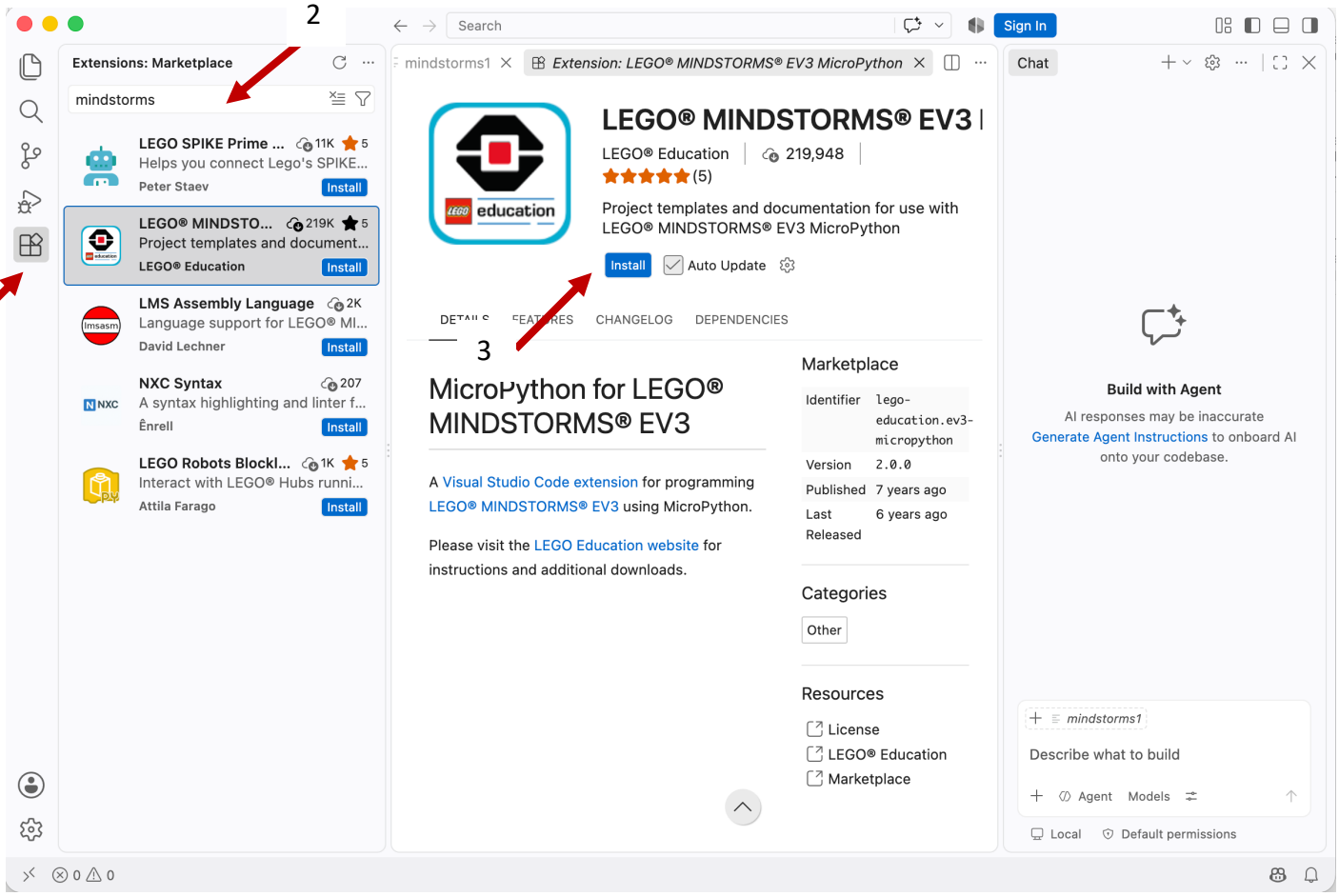
Visual Studio Code est un éditeur de code gratuit.

Il permet notamment de :

- écrire le programme
- organiser les fichiers
- repérer certaines erreurs
- envoyer le programme vers l'EV3

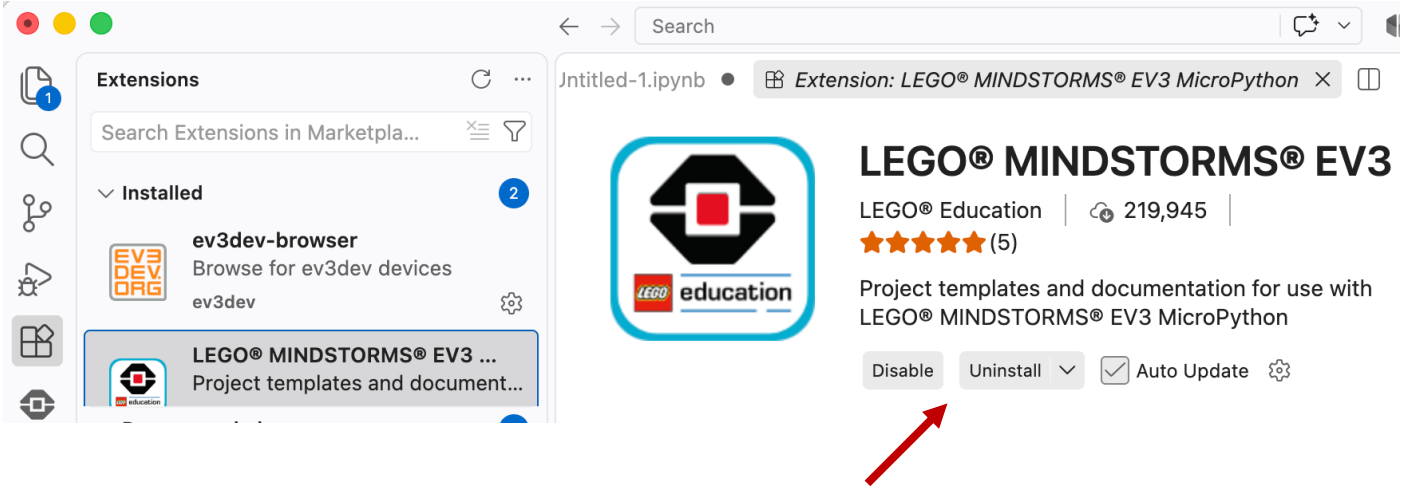
On peut le télécharger à l'adresse suivante : https://code.visualstudio.com/Download

Une fois installé et lancé, tu dois installer l'extension EV3 MicroPython



Une fois, l'extension installée, il faut vérifier qu'elle est bien activée.

Tu dois avoir ceci :



16. Préparation de la brique EV3



Pour pouvoir travailler en Python avec la brique EV3, il faut installer un nouveau firmware.

Pour garder le firmware d'origine intact, nous allons l'installer une carte microSDHC de maximum 32 Gb.

Tu peux télécharger le firmware ici : https://education.lego.com/en-us/product-resources/mindstorms-ev3/teacher-resources/python-for-ev3/

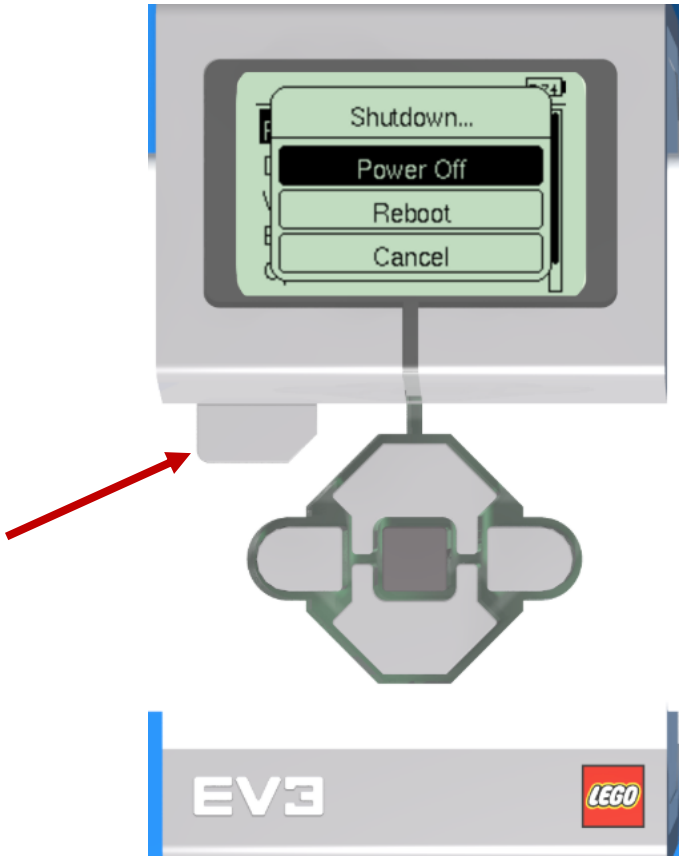
Ce firmware est basé sur pybricks, une bibliothèque créée par deux passionnés ; Laurens Valk (Pays-Bas) et David Lechner(USA), pour pouvoir utiliser MicroPython avec les briques intelligentes Lego.

Une fois la carte microSD flashée avec Etcher par exemple, il suffit de l'insérer dans la brique EV3, d'activer la brique et de la connecter à l'ordinateur exécutant Visual Studio.

Pour activer la brique EV3, il suffit d'appuyer sur le bouton central.



Pour l'éteindre, il faut appuyer sur le bouton Retour et sélectionner Power off dans le menu

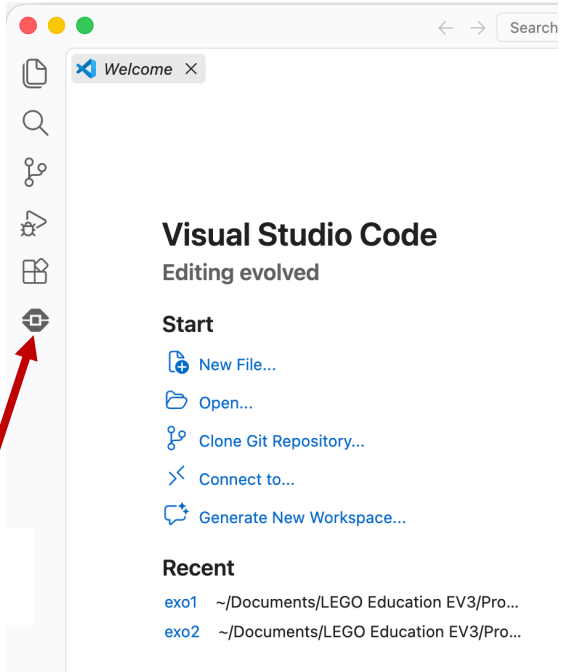


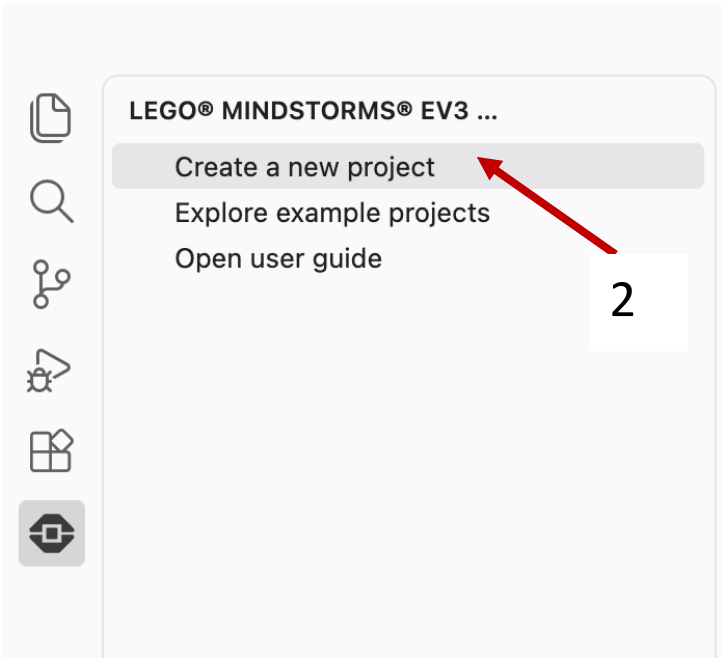
17. Utilisation de Visual Studio

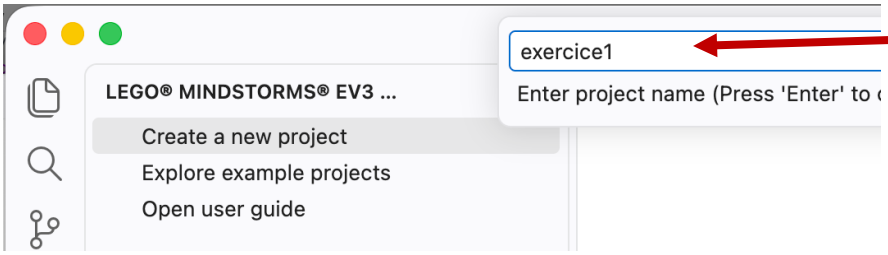
1. Démarre Visual Studio et, dans la barre d'activité, ouvre l'onglet **Extensions**.

2. Au niveau de l'onglet Lego Mindstorms EV3, crée un nouveau projet.

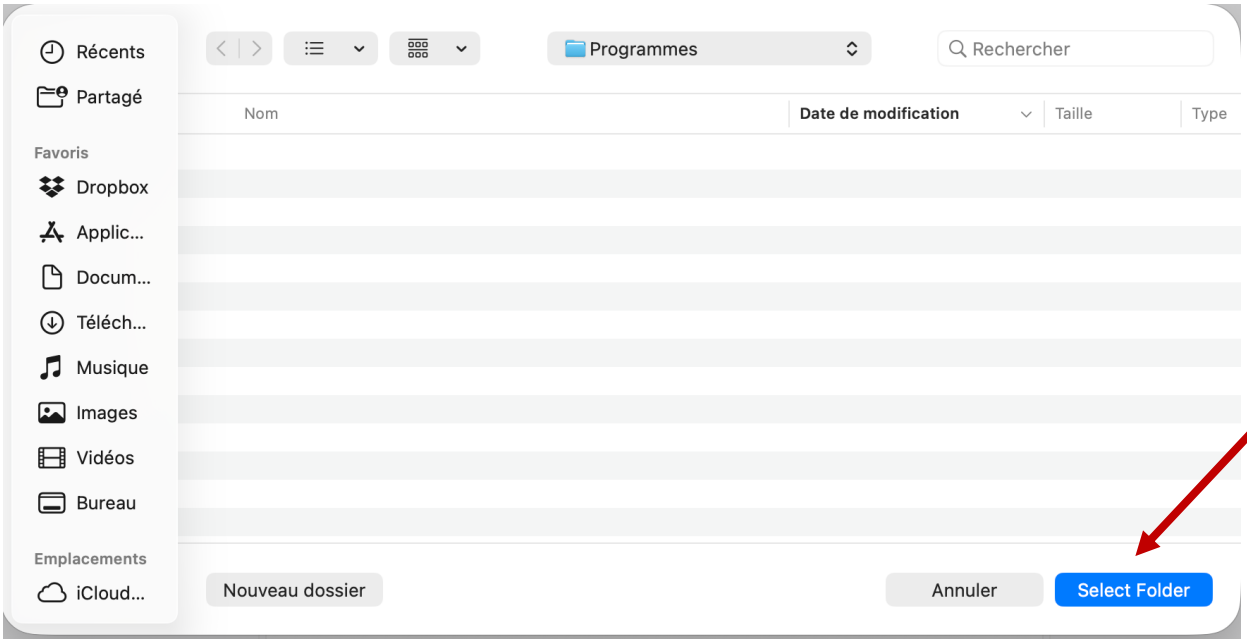
3. Sélectionne l'endroit sur le disque où se trouvera les fichiers du projet.






exercice1
Enter project name (Press 'Enter' to
LEGO® MINDSTORMS® EV3 ...
Create a new project
Explore example projects
Open user guide

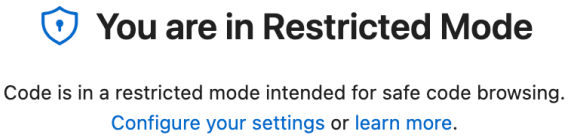
2

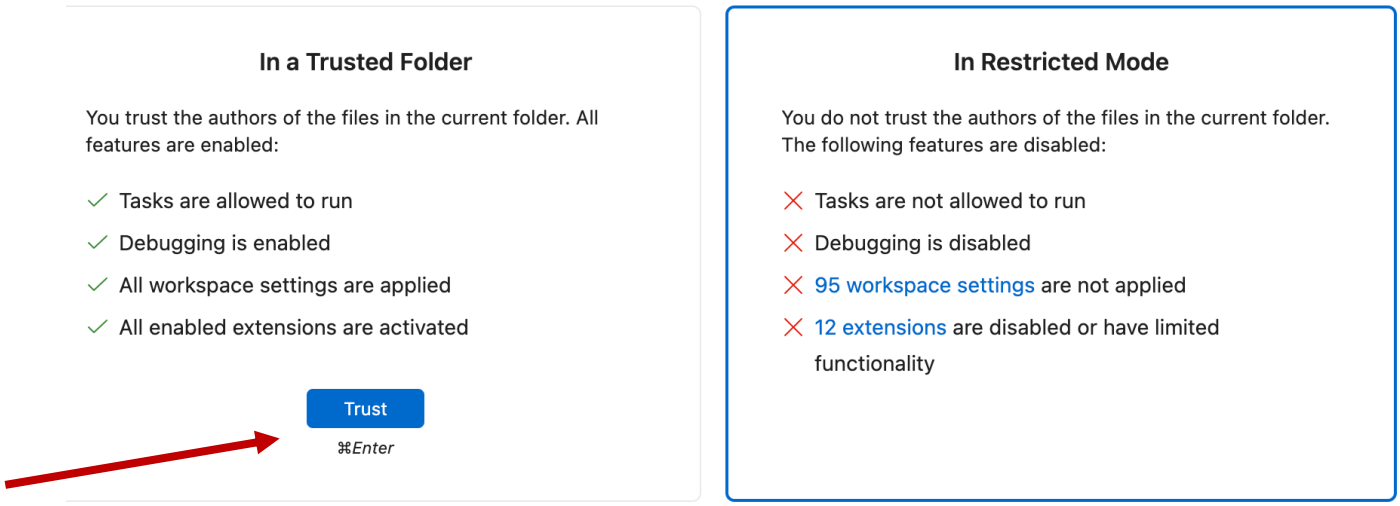

Récents
Partagé
Favoris
Dropbox
Applic...
Docum...
Téléch...
Musique
Images
Vidéos
Bureau
Emplacements
iCloud...
Programmes
Rechercher
Nom
Date de modification
Taille
Type
Nouveau dossier
Annuler
Select Folder

3

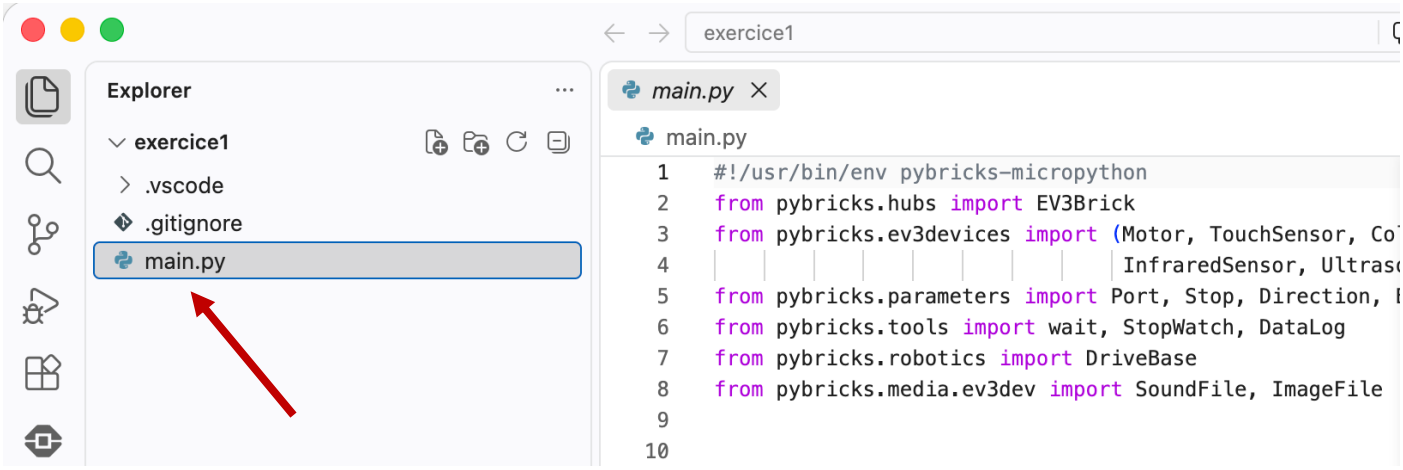
4. Passer du mode restreint au dossier de confiance. Vérifie que tu es bien dans l'onglet **Explorateur** au niveau de la barre d'activité.







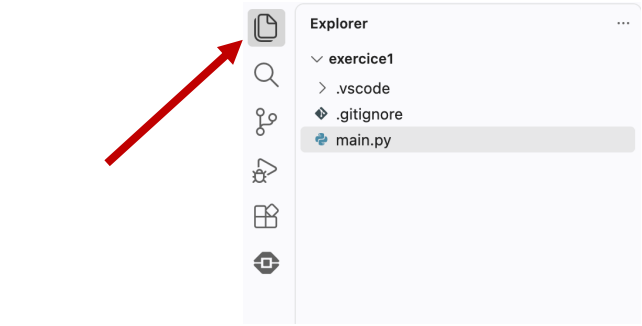
Le programme principal se trouve dans le fichier main.py

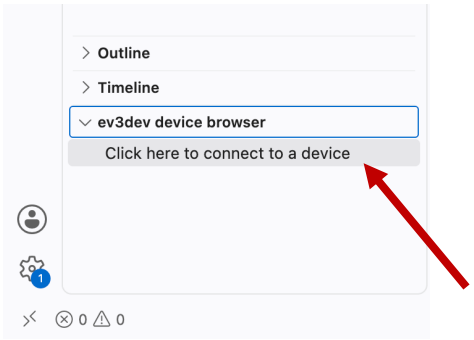


5. Pour connecter la brique EV3 à l'ordinateur, attends que MicroPython soit chargé (les led de la brique passent de l'orange clignotant au vert fixe).

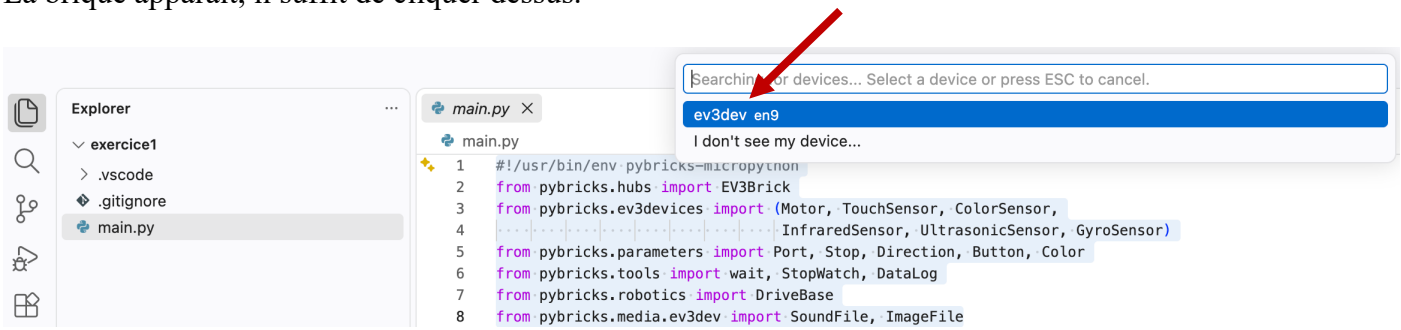
Connecte le câble USB à l'entrée PC de la brique.

Dans Visual Studio, au niveau de l'onglet **Explorateur**, clique sur **ev3dev device browser**, puis sur **Click here to connect to a device**.

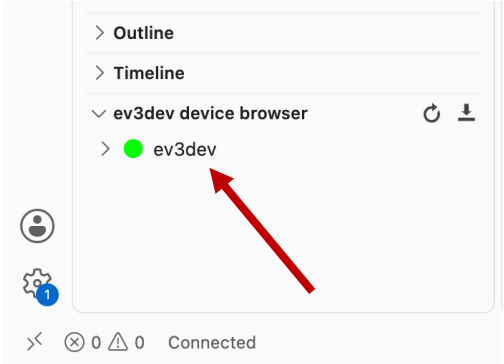




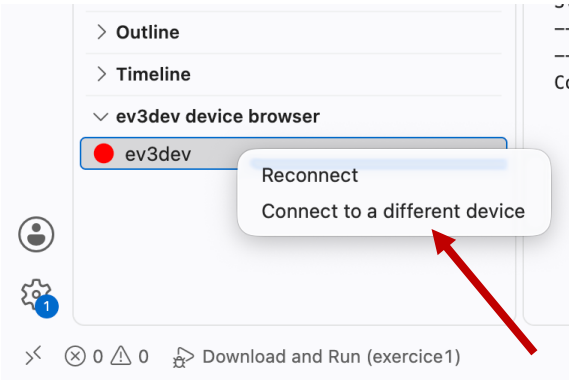
La brique apparaît, il suffit de cliquer dessus.



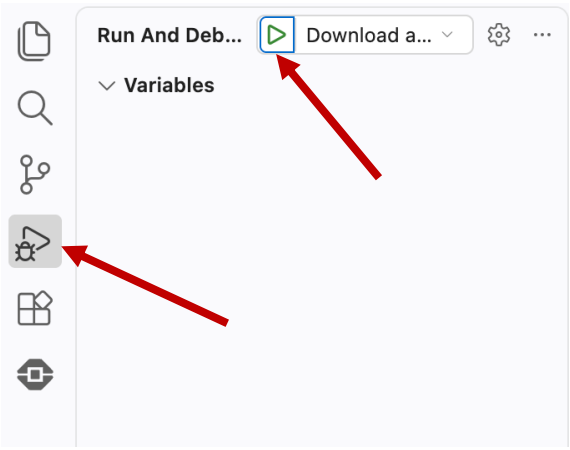
Elle est prête à l'emploi lorsqu'elle apparaît en vert



Si tu éteins et rallumes la brique EV3, le voyant passe au rouge et il faut cliquer avec le bouton droit de la souris sur **ev3dev** pour la reconnecter.



6. Pour charger et exécuter un programme sur la brique EV3, il faut passer à l'onglet **Exécuter et Déboguer**, de la barre d'activité, puis cliquer sur l'icône **Play**.



18. Quelques règles d'écriture de (Micro)Python

- Pas de point-virgule (;) à la fin de chaque ligne
- Pas d'accolades ({}) pour entourer un bloc de code
- C'est l'indentation (le retrait, généralement de 4 espaces) qui délimite le bloc de code
- Le bloc indenté sera toujours précédé par deux-points (:)
- (Micro)Python est sensible à la casse (il différencie majuscules et minuscules)
  `Limite`, `limite` et `LIMITE` représentent trois variables différentes
- Au niveau du nom des éléments :
  - Pour les variables et fonctions, on utilise le snake_case (mots en minuscules séparés par des tirets bas comme dans `limite_cuve`)
  - Pour les classes, on utilise le PascalCase (majuscule à chaque mot sans séparateur)
    Exemple : `class UserPremium`
  - Pour les constantes, tout en majuscule comme dans `PI = 3.14`
- Au niveau des espaces, on met un espace avant et après les opérateurs d'assignation et de comparaison (`x = 5 et pas x=5`).

  Par contre, pas d'espace à l'intérieur des parenthèses et crochets : `print(list[0])` et pas `print ( list [ 0 ] )`
- Pour écrire un commentaire sur une ligne, on utilise le symbole dièse (#)
  Exemple : `# Role de la fonction`

19. Comment commencer un programme

Tous les programmes doivent commencer par cette ligne :

```
#!/usr/bin/env pybricks-micropython
```

Cette ligne devra donc être rajoutée à tous les programmes de ce syllabus.

Si vous voulez pouvoir utiliser toutes les possibilités de la brique EV3 sans devoir y réfléchir spécifiquement par programme, commencez tous vos programmes par ce bloc :

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import (Motor, TouchSensor, ColorSensor,
                                 InfraredSensor, UltrasonicSensor, GyroSensor)
from pybricks.parameters import Port, Stop, Direction, Button, Color
from pybricks.tools import wait, StopWatch, DataLog
from pybricks.robotics import DriveBase
from pybricks.media.ev3dev import SoundFile, ImageFile
```

20. Notre premier programme

Nous allons commencer par faire produire un son à la brique EV3.

Créer un nouveau programme puis écrire :

```
from pybricks.hubs import EV3Brick

ev3 = EV3Brick()

ev3.speaker.beep()
```

Exécuter ensuite le programme sur la brique.

Si tout fonctionne correctement :

```
BIP
```

21. Comprendre la première ligne

```
from pybricks.hubs import EV3Brick
```

Cette instruction permet d'utiliser la brique EV3 dans notre programme.
Nous pouvons la lire approximativement comme :

```
Depuis la bibliothèque Pybricks, je veux utiliser EV3Brick
```

22. Qu'est-ce qu'une bibliothèque ?

Une bibliothèque contient du code déjà créé que nous pouvons réutiliser.
Sans bibliothèque, nous devrions programmer nous-mêmes tous les détails permettant de :

- faire fonctionner le haut-parleur
- contrôler un moteur
- lire un capteur
- afficher quelque chose

Pybricks nous fournit déjà ces outils.

23. Comprendre la deuxième ligne

```
ev3 = EV3Brick()
```

Cette ligne crée un objet représentant notre brique EV3.
Nous lui donnons le nom :

```
ev3
```

Nous pouvons ensuite utiliser ce nom pour commander la brique.

24. Comprendre la troisième ligne

```
ev3.speaker.beep()
```

Cette instruction demande :

```
à ev3
  ↓
d'utiliser son haut-parleur
  ↓
pour produire un bip
```

25. L'ordre des instructions

Python exécute généralement les instructions de haut en bas.

Par exemple :

```
print("A")
print("B")
print("C")
```

donne :

```
A
B
C
```

Le même principe s'applique au robot.

26. Faire plusieurs sons

Essayons :

```
from pybricks.hubs import EV3Brick

ev3 = EV3Brick()

ev3.speaker.beep()
ev3.speaker.beep()
ev3.speaker.beep()
```

Le robot reçoit trois instructions successives.

27. Problème

Les trois sons peuvent être très proches.

Nous voulons introduire une pause entre les instructions.

Pour cela, nous allons utiliser :

```
wait()
```

28. Importer `wait`

Nous devons ajouter :

```
from pybricks.tools import wait
```

Notre programme devient :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

ev3.speaker.beep()

wait(1000)

ev3.speaker.beep()
```

L'instruction `wait()` ne peut pas être utilisée si elle n'est pas importée au préalable.

29. Comprendre `wait(1000)`

`wait()` utilise des millisecondes.

Donc :

```
1000 ms = 1 seconde
```

Quelques conversions utiles :

```
100 ms = 0,1 seconde

500 ms = 0,5 seconde

1000 ms = 1 seconde

2000 ms = 2 secondes

5000 ms = 5 secondes
```

30. Exercice pratique 2

Créer un programme qui effectue :

```
BIP

attendre 2 secondes

BIP
```

Solution :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

ev3.speaker.beep()

wait(2000)

ev3.speaker.beep()
```

31. Exercice pratique 3

Créer :

```
BIP

attendre 500 ms

BIP

attendre 500 ms

BIP
```

Solution :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

ev3.speaker.beep()

wait(500)

ev3.speaker.beep()

wait(500)

ev3.speaker.beep()
```

32. Modifier un programme

Une partie importante de la programmation consiste à :

```
écrire
  ↓
tester
  ↓
observer
  ↓
modifier
  ↓
tester à nouveau
```

Nous n'écrivons pas nécessairement un programme parfait dès le premier essai.

33. Expérience

Tester :

```
wait(100)
```

Puis :

```
wait(500)
```

Puis :

```
wait(2000)
```

Observer l'effet sur le rythme des sons.

34. Ajouter un message

Nous pouvons utiliser Python pour afficher du texte dans la console.

Par exemple :

```
print("Bonjour EV3")
```

Programme :

```
from pybricks.hubs import EV3Brick

ev3 = EV3Brick()

print("Bonjour EV3")

ev3.speaker.beep()
```

35. Pourquoi utiliser `print()` ?

`print()` est très utile pour :

- `afficher une information`
- `tester un programme`
- `observer une valeur`
- `déboguer`

Nous l'utiliserons très souvent pendant les prochains cours.

36. Exercice pratique 4

Créer un programme qui affiche :

```
Démarrage
```

puis fait un son.

Attend une seconde.

Puis affiche :

```
Terminé
```

Solution :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

print("Démarrage")

ev3.speaker.beep()

wait(1000)

print("Terminé")
```

37. Les commentaires

Nous pouvons écrire des informations destinées aux humains dans le programme.
Un commentaire commence par :

```
#
```

Exemple :

```
# Création de la brique EV3
ev3 = EV3Brick()
```

Python ignore le commentaire.

38. Pourquoi utiliser des commentaires ?

Ils permettent d'expliquer :

- le rôle d'une partie du programme
- un choix important
- une étape du fonctionnement

Exemple :

```
# Faire un son au démarrage
ev3.speaker.beep()
```

39. Attention aux commentaires inutiles

Ceci est peu utile :

```
# Faire beep
ev3.speaker.beep()
```

Le code est déjà très clair.

Un commentaire est plus intéressant lorsqu'il apporte une information supplémentaire.

40. Première structure de programme

Nous pouvons déjà organiser notre code.

```
# IMPORTS

from pybricks.hubs import EV3Brick
from pybricks.tools import wait


# INITIALISATION

ev3 = EV3Brick()


# PROGRAMME

print("Début")

ev3.speaker.beep()

wait(1000)

ev3.speaker.beep()

print("Fin")
```

Cette organisation deviendra de plus en plus importante lorsque les programmes deviendront complexes.

41. Qu’est-ce qu’une erreur ?

Une erreur peut empêcher le programme de fonctionner.

Exemple incorrect :

```
ev3.speaker.beep(
```

Il manque :

```
)
```

Python ne pourra pas correctement comprendre l’instruction.

42. Autre exemple

Incorrect :

```
ev3 speaker beep()
```

Correct :

```
ev3.speaker.beep()
```

La syntaxe doit être respectée.

43. Les parenthèses

Nous rencontrons déjà plusieurs exemples :

```
EV3Brick()
beep()
wait(1000)
```

Les parenthèses indiquent généralement que nous appelons une fonction ou une méthode.

Nous approfondirons cette notion plus tard.

44. Les paramètres

Comparer :

```
wait(500)
```

et :

```
wait(2000)
```

Nous utilisons la même fonction :

```
wait
```

mais avec des valeurs différentes.

La valeur placée entre parenthèses influence le comportement.

Nous appellerons plus tard cela un paramètre ou un argument selon le contexte.

45. Exercice pratique 5

Sans copier la solution précédente, créer un programme qui réalise :

```
Afficher : Robot prêt

BIP

attendre 1 seconde

BIP

attendre 2 secondes

BIP

Afficher : Fin du programme
```

46. Solution possible

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

print("Robot prêt")

ev3.speaker.beep()
wait(1000)

ev3.speaker.beep()
wait(2000)

ev3.speaker.beep()

print("Fin du programme")
```

47. Mini défi 1 : compte à rebours manuel

Créer un programme qui affiche :

```
3

2

1

GO
```

avec une seconde entre chaque affichage.

Solution :

```
from pybricks.tools import wait

print(3)

wait(1000)

print(2)

wait(1000)

print(1)

wait(1000)

print("GO")
```

Plus tard, nous apprendrons à faire cela beaucoup plus simplement avec une boucle.

48. Mini défi 2 : signal de départ

Créer :

```
BIP

pause

BIP

pause

BIP

pause plus longue

BIP
```

Choisis les valeurs de `wait()`.

L'objectif est d'expérimenter.

49. Mini défi 3 : message et son

Créer un petit scénario :

```
Initialisation
↓
BIP
↓
Attente
↓
Robot prêt
↓
BIP
```

Tu dois écrire le programme sans solution fournie au départ.

50. Première notion importante

Un programme est une suite d'instructions précises.

Le robot ne devine pas ce que nous voulons.

Si nous écrivons :

```
wait(100)
```

il attend 100 millisecondes.

Il ne sait pas que nous pensions :

```
100 secondes
```

La précision est donc essentielle en programmation.

51. Deuxième notion importante

Le robot fait exactement ce que le programme lui demande.

Pas nécessairement ce que le programmeur voulait lui demander.

Cette différence sera à l'origine de nombreux bugs.

52. Troisième notion importante

Nous pouvons améliorer un programme progressivement.

```
VERSION 1

faire un bip
```

Puis :

```
VERSION 2

faire deux bips
```

Puis :

```
VERSION 3

ajouter des pauses
```

Puis :

```
VERSION 4

ajouter des messages
```

Cette méthode sera utilisée pendant toute la formation.

53. Défi final : séquence de démarrage EV3

Créer une séquence de démarrage complète.

Contraintes :

1. afficher `Initialisation`
2. faire un bip
3. attendre 500 ms
4. afficher `Test système`
5. faire deux bips séparés de 300 ms
6. attendre une seconde
7. afficher `Robot prêt`
8. faire un dernier bip

54. Exemple de solution

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

print("Initialisation")

ev3.speaker.beep()

wait(500)

print("Test système")

ev3.speaker.beep()

wait(300)

ev3.speaker.beep()

wait(1000)

print("Robot prêt")

ev3.speaker.beep()
```

55. Questions de fin de cours

Question 1
Quel est le rôle d'un capteur ?

Question 2
Quel est le rôle d'un moteur ?

Question 3
Quel est le rôle du programme ?

Question 4
Quels sont les ports utilisés principalement pour les moteurs ?

Question 5
Quels sont les ports utilisés principalement pour les capteurs ?

Question 6
Qu'est-ce que Python ?

Question 7
Qu'est-ce que MicroPython ?

Question 8
À quoi sert :
`ev3.speaker.beep()`

Question 9
Que représente :
`wait(1000)`

Question 10
Dans quel ordre les instructions sont elles normalement exécutées ?

Question 11
À quoi sert `print()` ?

Question 12
À quoi sert `#` en Python ?

Correction

Réponse 1
À fournir des informations sur l'environnement au programme.

Réponse 2
À produire une action physique.

Réponse 3
À définir les instructions et les décisions du robot.

Réponse 4
```
A
B
C
D
```

Réponse 5
```
1
2
3
4
```

Réponse 6
Un langage de programmation.

Réponse 7
Une version de Python adaptée aux petits systèmes électroniques et embarqués.

Réponse 8
À faire produire un son à la brique EV3.

Réponse 9
Une attente de :
```
1000 millisecondes
=
1 seconde
```

Réponse 10
De haut en bas, sauf lorsqu'une structure de programmation modifie cet ordre.

Réponse 11
À afficher une information.

Réponse 12
À écrire un commentaire ignoré par Python.

56. Objectifs à atteindre

- pouvoir prédire ce que va faire un programme avant de l'exécuter
- pouvoir le modifier facilement
- ne pas confondre millisecondes et secondes
- ne pas oublier les imports

57. Méthode de débogage

Ne pas modifier plusieurs éléments alors que le programme ne fonctionne pas.

La bonne méthode :

```
modifier une chose
  ↓
tester
  ↓
observer
```

Cette habitude sera très importante pour les projets futurs.

Cours 2 : Les variables et les types en MicroPython avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

- comprendre ce qu'est une variable
- créer une variable en Python
- modifier la valeur d'une variable
- reconnaître les principaux types de données
- utiliser des variables pour contrôler un robot EV3
- comprendre pourquoi les variables rendent un programme plus facile à modifier

Matériel nécessaire

- 1 LEGO Education Mindstorms EV3
- 1 moteur EV3
- 1 ordinateur avec Visual Studio Code
- MicroPython installé sur la carte microSD de l'EV3
- câble USB ou connexion permettant de transférer le programme

1. Rappel du cours précédent

Lors du premier cours, nous avons vu qu'un programme est une suite d'instructions.

Exemple :

```
ev3.speaker.beep()
```

Cette instruction demande à la brique EV3 de produire un son.

Nous avons également utilisé :

```
wait(1000)
```

Cela signifie que le programme attend pendant 1000 millisecondes.

Donc :

```
1000 millisecondes = 1 seconde
```

Aujourd'hui, nous allons apprendre à stocker des informations dans notre programme.

2. Qu’est-ce qu’une variable ?

Une variable peut être imaginée comme une boîte dans laquelle nous pouvons placer une information.

Par exemple :

```
age = 18
```

Nous avons une boîte appelée :

```
age
```

et cette boîte contient :

```
18
```

On peut représenter cela ainsi :

```
age
 ↓
18
```

Autre exemple :

```
vitesse = 500
```

La variable `vitesse` contient maintenant la valeur `500`.

3. Créer une variable

Pour créer une variable en Python, on écrit :

```
nom = valeur
```

Exemple :

```
vitesse = 500
```

On peut ensuite afficher son contenu :

```
vitesse = 500

print(vitesse)
```

Le programme affiche :

```
500
```

Exercice 1

Créer une variable appelée :

```
distance
```

et lui donner la valeur :

```
100
```

Solution :

```
distance = 100

print(distance)
```

4. Modifier une variable

Une variable peut changer pendant l'exécution du programme.

Exemple :

```
vitesse = 200

print(vitesse)

vitesse = 500

print(vitesse)
```

Le programme affiche :

```
200
500
```

La deuxième valeur remplace donc la première.

Exercice 2

Que va afficher ce programme ?

```
nombre = 10

print(nombre)

nombre = 20

print(nombre)

nombre = 50

print(nombre)
```

Réponse :

```
10
20
50
```

5. Pourquoi utiliser des variables ?

Imaginons ce programme :

```
moteur.run(500)
wait(2000)

moteur.stop()
```

Si nous voulons changer la vitesse, nous devons modifier directement la valeur `500`.

Nous pouvons plutôt écrire :

```
vitesse = 500

moteur.run(vitesse)
wait(2000)

moteur.stop()
```

Maintenant, pour modifier la vitesse du robot, nous pouvons simplement modifier :

```
vitesse = 500
```

Par exemple :

```
vitesse = 800
```

Cela rend le programme plus facile à lire et à modifier.

6. Premier programme avec un moteur

Connecter un moteur sur le port A.



Créer ensuite le programme suivant :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 500

moteur.run(vitesse)

wait(2000)

moteur.stop()
```

Que fait le programme ?

Il crée d'abord une variable :

```
vitesse = 500
```

Puis il utilise cette variable :

```
moteur.run(vitesse)
```

Le moteur tourne donc à :

```
500 degrés par seconde
```

Le programme attend ensuite :

```
wait(2000)
```

soit :

```
2 secondes
```

Puis le moteur s'arrête.

7. Deux variables

Nous pouvons également stocker la durée dans une variable.

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 500
duree = 2000

moteur.run(vitesse)

wait(duree)

moteur.stop()
```

Nous pouvons maintenant contrôler le comportement du moteur uniquement avec :

```
vitesse = 500
duree = 2000
```

Exercice 3

Modifier uniquement les variables pour que le moteur tourne :

- à 300 degrés par seconde
- pendant 3 secondes

Solution :

```
vitesse = 300
duree = 3000
```

8. Les principaux types de variables

Toutes les variables ne contiennent pas forcément des nombres.

Python peut stocker différents types d'informations.

Nombre entier

```
vitesse = 500
```
`500` est un nombre entier.

En Python, ce type est appelé :

```
int
```

Nombre décimal

```
distance = 25.5
```
`25.5` est un nombre décimal.

Ce type est appelé :

```
float
```

Texte

```
nom = "EV3"
```

Le texte est placé entre guillemets.

Ce type est appelé :

```
str
```

Valeur vraie ou fausse

```
robot_actif = True
```

Une variable peut également contenir :

`True`
ou :
`False`

Ce type est appelé :

```
bool
```

9. Observer le type d’une variable

Python possède une fonction appelée :

```
type()
```

Exemple :

```
vitesse = 500

print(type(vitesse))
```

Le programme indique que `vitesse` contient un nombre entier.

Essayons maintenant :

```
nom = "EV3"

print(type(nom))
```

Puis :

```
distance = 20.5

print(type(distance))
```

10. Utiliser plusieurs variables

Créons maintenant plusieurs réglages pour notre moteur.

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse_lente = 200
vitesse_rapide = 800
duree = 2000

moteur.run(vitesse_lente)
wait(duree)

moteur.run(vitesse_rapide)
wait(duree)

moteur.stop()
```

Le moteur tourne d’abord lentement puis rapidement.

11. Exercice pratique

Créer un programme qui effectue les actions suivantes :

```
Moteur lent pendant 2 secondes

Puis

Moteur rapide pendant 3 secondes

Puis

Arrêt
```

Les vitesses doivent être stockées dans des variables.

Exemple :

```
vitesse_lente = 200
vitesse_rapide = 800
```

12. Faire tourner un moteur dans l'autre sens

Une vitesse positive fait tourner le moteur dans un sens.

Par exemple :

```
vitesse = 500
```

Une vitesse négative permet de le faire tourner dans l'autre sens.

```
vitesse = -500
```

Essayons :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 500

moteur.run(vitesse)
wait(2000)

moteur.run(-vitesse)
wait(2000)

moteur.stop()
```

Le moteur tourne pendant deux secondes dans un sens puis pendant deux secondes dans l'autre sens.

13. Faire des calculs avec des variables

Les variables peuvent également être utilisées dans des calculs.

Exemple :

```
vitesse = 200

vitesse_rapide = vitesse * 3

print(vitesse_rapide)
```

Résultat :

```
600
```

Nous pouvons utiliser ce principe avec le moteur :

```
vitesse = 200

moteur.run(vitesse)

wait(2000)

moteur.run(vitesse * 3)

wait(2000)

moteur.stop()
```

14. Les opérateurs mathématiques

Python permet d'effectuer différents calculs.

Addition :
```
a = 10 + 5
```

Soustraction :
```
a = 10 - 5
```

Multiplication :
```
a = 10 * 5
```

Division :
```
a = 10 / 5
```

Exemple :

```
vitesse = 200

vitesse_double = vitesse * 2

print(vitesse_double)
```

Résultat :

```
400
```

15. Exercice de réflexion

On possède :

```
vitesse = 250
```

Que contiendront les variables suivantes ?

```
vitesse2 = vitesse * 2

vitesse3 = vitesse + 100

vitesse4 = vitesse / 2
```

Réponses :

```
vitesse2 = 500

vitesse3 = 350

vitesse4 = 125
```

16. Défi pratique 1

Programmer le moteur pour effectuer la séquence suivante :

```
200 degrés par seconde pendant 2 secondes

400 degrés par seconde pendant 2 secondes

800 degrés par seconde pendant 2 secondes

Arrêt
```

Il faut utiliser au minimum trois variables.

17. Défi pratique 2

Créer seulement une variable de départ :

```
vitesse = 200
```

Le programme doit ensuite :

```
tourner à vitesse

puis

tourner à vitesse multipliée par 2

puis

tourner à vitesse multipliée par 4
```

Exemple attendu :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 200
duree = 2000

moteur.run(vitesse)
wait(duree)

moteur.run(vitesse * 2)
wait(duree)

moteur.run(vitesse * 4)
wait(duree)

moteur.stop()
```

18. Défi pratique 3

Faire tourner le moteur :

```
2 secondes vers l'avant

1 seconde à l'arrêt

2 secondes vers l'arrière

puis arrêt définitif
```

La vitesse et les durées doivent être stockées dans des variables.

19. Erreurs fréquentes

Erreur 1

Écrire :

```
vitesse == 500
```

alors que l'on veut créer une variable.

Pour créer ou modifier une variable, on utilise :

```
vitesse = 500
```

Nous verrons plus tard que == possède une autre signification.

Erreur 2

Utiliser une variable qui n'existe pas.

Exemple :

```
moteur.run(vitese)
```

alors que la variable créée était :

```
vitesse = 500
```

Python considère `vitesse` et `vitese` comme deux noms différents.

Erreur 3

Oublier les guillemets pour du texte.

Incorrect :

```
nom = EV3
```

Correct :

```
nom = "EV3"
```

20. Bonnes pratiques

Il est important de donner des noms compréhensibles aux variables.

Éviter :

```
x = 500
y = 2000
```

Préférer :

```
vitesse = 500
duree = 2000
```

Le programme devient beaucoup plus facile à comprendre.

Pour un robot, nous pourrons par exemple utiliser :

```
vitesse = 500
distance = 300
angle = 90
duree = 2000
```

21. Mini projet de fin de cours

Le moteur à trois vitesses

Ton programme doit respecter les étapes suivantes :

```
Démarrage
   ↓
Tourner lentement pendant 2 secondes
   ↓
Tourner à vitesse moyenne pendant 2 secondes
   ↓
Tourner rapidement pendant 2 secondes
   ↓
Arrêt pendant 1 seconde
   ↓
Tourner rapidement dans l'autre sens pendant 2 secondes
   ↓
Arrêt
```

Tu dois définir les variables au début du programme.

Par exemple :

```
vitesse_lente = 200
vitesse_moyenne = 400
vitesse_rapide = 800

duree = 2000
pause = 1000
```

Une solution possible

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse_lente = 200
vitesse_moyenne = 400
vitesse_rapide = 800

duree = 2000
pause = 1000

moteur.run(vitesse_lente)
wait(duree)

moteur.run(vitesse_moyenne)
wait(duree)

moteur.run(vitesse_rapide)
wait(duree)

moteur.stop()
wait(pause)

moteur.run(-vitesse_rapide)
wait(duree)
moteur.stop()
```

22. Questions de fin de cours

Question 1

Qu'est ce qu'une variable ?

Question 2

Que signifie :

```
vitesse = 500
```

Question 3

Quelle est la différence entre :

```
vitesse = 500
```

et :

```
nom = "EV3"
```

Question 4

Quel est le résultat de :

```
vitesse = 300
nouvelle_vitesse = vitesse * 2
```

Question 5

Comment faire tourner un moteur dans le sens inverse ?

Correction

Réponse 1

Une variable permet de stocker une information dans le programme.

Réponse 2

Une variable appelée `vitesse` est créée et reçoit la valeur `500`.

Réponse 3

`500` est un nombre entier alors que `"EV3"` est du texte.

Réponse 4

```
600
```

Réponse 5

On peut utiliser une vitesse négative.

Par exemple :

```
moteur.run(-500)
```

Cours 3 : Calculs, opérateurs et déplacements avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

- utiliser les principaux opérateurs mathématiques en Python
- effectuer des calculs avec des variables
- comprendre la différence entre distance, vitesse et durée
- calculer le périmètre d'une roue
- estimer le nombre de rotations nécessaires pour parcourir une distance
- utiliser ces calculs pour commander un moteur EV3

Matériel nécessaire

- 1 LEGO Education Mindstorms EV3
- 1 moteur EV3
- 1 roue LEGO
- 1 ordinateur avec Visual Studio Code
- MicroPython installé sur l'EV3
- une règle ou un mètre
- une calculatrice facultative

1. Rappel du cours précédent

Lors du cours précédent, nous avons appris à créer des variables.

Exemple :

```
vitesse = 500
duree = 2000
```

Nous avons également vu qu'une variable peut être utilisée dans un calcul.

```
vitesse = 200

vitesse_double = vitesse * 2

print(vitesse_double)
```

Résultat :

```
400
```

Aujourd'hui, nous allons utiliser ces calculs pour contrôler plus précisément notre robot.

2. Les principaux opérateurs mathématiques

Python permet d'effectuer les opérations mathématiques classiques.

Addition

```
resultat = 10 + 5
```

Résultat :

```
15
```

Soustraction

```
resultat = 10 - 5
```

Résultat :

```
5
```

Multiplication

```
resultat = 10 * 5
```

Résultat :

```
50
```

Division

```
resultat = 10 / 5
```

Résultat :

```
2
```

3. Calculer avec des variables

Il est possible d'utiliser directement des variables dans une opération.

```
a = 10
b = 5

resultat = a + b

print(resultat)
```

Le programme affiche :

```
15
```

Exercice 1

Complète mentalement les résultats.

```
a = 20
b = 4

addition = a + b
soustraction = a - b
multiplication = a * b
division = a / b
```

Résultats :

```
addition = 24
soustraction = 16
multiplication = 80
division = 5
```

4. Modifier une variable avec un calcul

On peut utiliser la valeur actuelle d'une variable pour calculer sa nouvelle valeur.

```
vitesse = 200

vitesse = vitesse + 100

print(vitesse)
```

Résultat :

```
300
```

Le programme effectue d'abord :

```
200 + 100
```

puis remplace l'ancienne valeur de `vitesse` par :

```
300
```

Exercice 2

Que vaut `distance` à la fin ?

```
distance = 100

distance = distance + 50

distance = distance * 2
```

Réponse :

```
300
```

5. Ordre des opérations

Python respecte les règles mathématiques habituelles.

Par exemple :

```
resultat = 10 + 5 * 2
```

Python effectue d'abord :

```
5 × 2
```

puis :

```
10 + 10
```

Résultat :

```
20
```

Pour changer l'ordre, on peut utiliser des parenthèses.

```
resultat = (10 + 5) * 2
```

Résultat :

```
30
```

6. Exercice sur les parenthèses

Quelle est la différence entre :

```
a = 10 + 5 * 2
```

et :

```
b = (10 + 5) * 2
```

Réponse :

```
a = 20
b = 30
```

7. Distance, vitesse et durée

En robotique, trois grandeurs reviennent très souvent :

```
Distance

Vitesse

Durée
```

On peut utiliser la relation :

```
Distance = Vitesse × Temps
```

Par exemple, si un robot avance à :

```
20 cm/s
```

pendant :

```
5 secondes
```

alors il parcourt :

```
20 × 5 = 100 cm
```

8. Calcul en Python

```
vitesse = 20
temps = 5

distance = vitesse * temps

print(distance)
```

Résultat :

```
100
```

Exercice 3

Un robot avance à :

```
30 cm/s
```

pendant :

```
4 secondes
```

Quelle distance parcourt il ?

Calcul :

```
30 × 4 = 120 cm
```

9. Trouver le temps

Nous pouvons également utiliser :

```
Temps = Distance / Vitesse
```

Exemple :

Un robot doit parcourir :

```
200 cm
```

à :

```
50 cm/s
```

Calcul :

```
200 / 50 = 4 secondes
```

En Python :

```
distance = 200
vitesse = 50

temps = distance / vitesse

print(temps)
```

10. Application avec un moteur

Un moteur EV3 peut tourner à une certaine vitesse exprimée en degrés par seconde.

Exemple :

```
moteur.run(360)
```

Cela signifie que le moteur tourne à :

```
360 degrés par seconde
```

Une rotation complète correspond à :

```
360 degrés
```

Donc à 360 degrés par seconde, le moteur effectue environ :

```
1 rotation par seconde
```

11. Calculer une durée de rotation

Supposons que nous voulions faire tourner le moteur de :

```
720 degrés
```

avec une vitesse de :

```
360 degrés par seconde
```

Nous pouvons calculer :

```
temps = angle / vitesse
```

Donc :

```
720 / 360 = 2 secondes
```

En Python :

```
angle = 720
vitesse = 360

temps = angle / vitesse

print(temps)
```

Attention : `wait()` utilise des millisecondes.

Il faut donc convertir les secondes en millisecondes.

```
temps_ms = temps * 1000
```

12. Programme complet

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

angle = 720
vitesse = 360

temps = angle / vitesse
temps_ms = temps * 1000

moteur.run(vitesse)

wait(temps_ms)

moteur.stop()
```

13. Une solution plus précise

Pybricks permet également de demander directement au moteur de tourner d'un certain angle.

```
moteur.run_angle(360, 720)
```

Ici :

```
360
```

correspond à la vitesse.

Et :

```
720
```

correspond à l'angle.

Cela signifie :

```
faire tourner le moteur de 720 degrés
à 360 degrés par seconde
```

14. Comparaison des deux méthodes

Première méthode :

```
moteur.run(vitesse)
wait(duree)
moteur.stop()
```

Cette méthode utilise principalement le temps.

Deuxième méthode :

```
moteur.run_angle(vitesse, angle)
```

Cette méthode utilise directement l'angle du moteur.

Pour des déplacements précis, la deuxième méthode est souvent plus adaptée.

15. Comprendre la rotation d'une roue

Une roue effectue une rotation complète lorsque le moteur tourne de :

```
360 degrés
```

Si la roue mesure 56 mm de diamètre, elle ne parcourt pas 56 mm en une rotation.

Elle parcourt son périmètre.

La formule est :

```
Périmètre = π × diamètre
```

Avec :

```
π ≈ 3,14
```

Pour une roue de 56 mm :

```
3,14 × 56
```

donne environ :

```
175,84 mm
```

Donc une rotation complète permet au robot d'avancer d'environ :

```
17,6 cm
```

16. Calcul en Python

```
pi = 3.14
diametre = 56

perimetre = pi * diametre

print(perimetre)
```

Résultat approximatif :

```
175.84
```

17. Combien de rotations pour parcourir 1 mètre ?

Nous savons qu'une rotation parcourt environ :

```
175,84 mm
```

Nous voulons parcourir :

```
1000 mm
```

Il faut donc calculer :

```
1000 / 175,84
```

Résultat approximatif :

```
5,69 rotations
```

18. Transformer les rotations en degrés

Une rotation correspond à :

```
360 degrés
```

Donc :

```
5,69 × 360
```

donne environ :

```
2048 degrés
```

Le moteur devra donc tourner d'environ 2048 degrés.

19. Programme pour parcourir environ 1 mètre

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

diametre = 56
distance = 1000

perimetre = 3.14 * diametre

rotations = distance / perimetre

angle = rotations * 360

moteur.run_angle(500, angle)
```

20. Exercice pratique

Calcule le nombre de rotations nécessaires pour parcourir :

```
500 mm
```

avec une roue de :

```
56 mm
```

Calcul :

```
Périmètre = 3,14 × 56
```

soit environ :

```
175,84 mm
```

Puis :

```
500 / 175,84
```

soit environ :

```
2,84 rotations
```

Enfin :

```
2,84 × 360
```

soit environ :

```
1022 degrés
```

21. Tester dans la réalité

Le calcul théorique ne sera pas toujours parfaitement exact.

Pourquoi ?

Parce que plusieurs éléments peuvent influencer le déplacement :

- le diamètre réel des roues
- le poids du robot
- le niveau de batterie
- le sol
- le glissement des roues
- la vitesse du moteur

La robotique nécessite donc souvent :

```
calcul

puis

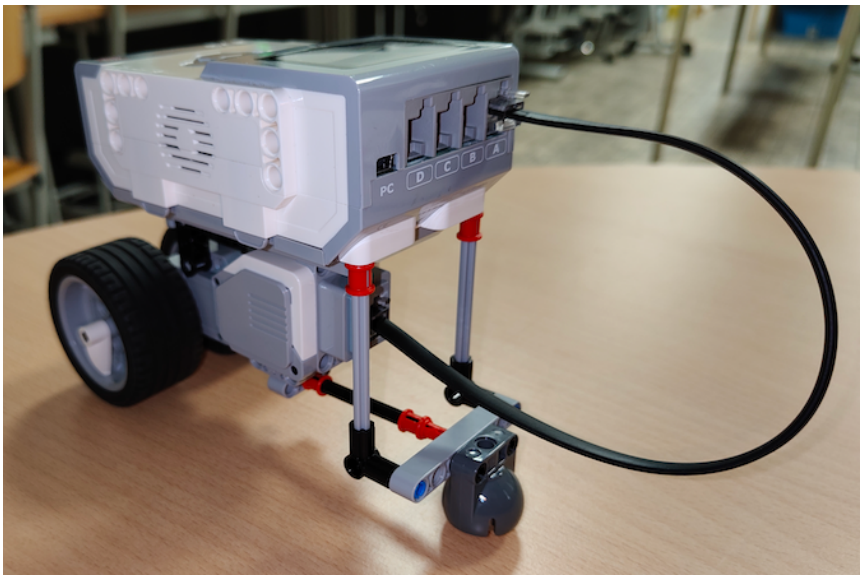
test

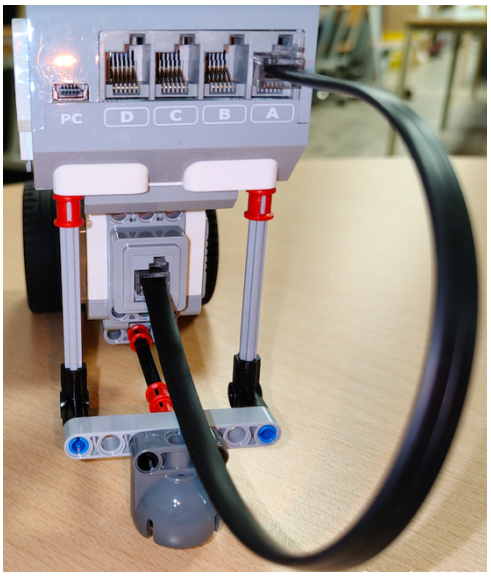
puis

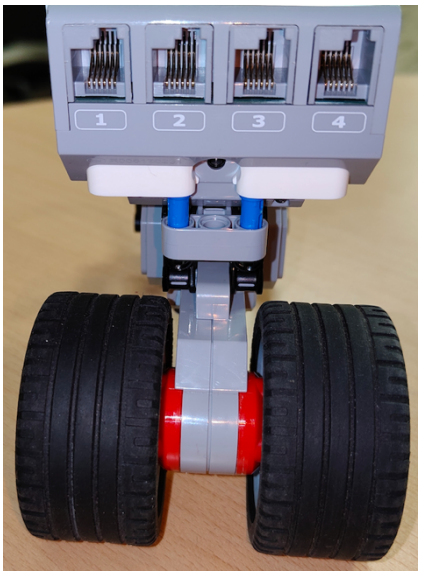
correction
```

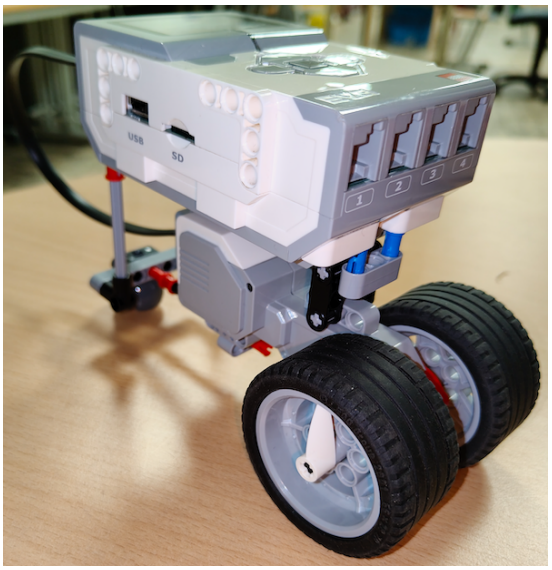
22. Expérience

Construis un robot simple avec deux roues reliées au même moteur, connecté au port A de la brique ev3 placée au-dessus. Une roulette à bille sera placée à l'arrière.









Place le robot au sol.

Demande lui de parcourir théoriquement :

`1000 mm`

Mesure ensuite la distance réelle.

Note les résultats.

Essai	Distance demandée	Distance réelle
1	1000 mm	
2	1000 mm	
3	1000 mm	

Calcule ensuite la moyenne.

23. Comprendre l’erreur

Si le robot devait parcourir :

```
1000 mm
```

mais qu’il parcourt :

```
950 mm
```

l’erreur est :

```
1000 - 950
```

donc :

```
50 mm
```

Nous pouvons créer :

```
distance_voulue = 1000
distance_reelle = 950

erreur = distance_voulue - distance_reelle

print(erreur)
```

Résultat :

```
50
```

Cette notion d’erreur deviendra très importante plus tard lorsque nous travaillerons avec les capteurs et le suivi de ligne.

24. Mini défi 1

Programme un moteur pour effectuer exactement :

```
3 rotations
```

avec une vitesse de :

```
500 degrés par seconde
```

Il faut d’abord calculer l’angle total.

```
3 × 360 = 1080 degrés
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

rotations = 3
angle = rotations * 360

moteur.run_angle(500, angle)
```

25. Mini défi 2

Le diamètre de la roue est :

```
56 mm
```

Le robot doit parcourir :

```
750 mm
```

Créer un programme qui calcule automatiquement l'angle nécessaire.

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

diametre = 56
distance = 750
pi = 3.14

perimetre = pi * diametre

rotations = distance / perimetre
angle = rotations * 360

moteur.run_angle(500, angle)
```

26. Mini défi 3

Modifier le programme précédent pour pouvoir changer uniquement cette variable :

```
distance = 750
```

et faire fonctionner le programme pour :

```
250 mm

500 mm

750 mm

1000 mm
```

L'objectif est de comprendre qu'un bon programme peut être réutilisé sans modifier toute sa logique.

27. Défi final

Créer un programme qui commande un moteur afin de simuler ce parcours :

```
Avancer de 500 mm

Puis revenir de 250 mm

Puis avancer de 1000 mm
```

Utiliser une variable pour le diamètre de roue.

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

diametre = 56
pi = 3.14
vitesse = 500

perimetre = pi * diametre

distance1 = 500
distance2 = 250
distance3 = 1000

angle1 = distance1 / perimetre * 360
angle2 = distance2 / perimetre * 360
angle3 = distance3 / perimetre * 360

moteur.run_angle(vitesse, angle1)

moteur.run_angle(vitesse, -angle2)

moteur.run_angle(vitesse, angle3)
```

28. Questions de fin de cours

Question 1

Quel symbole permet de faire une multiplication en Python ?

Question 2

Quel symbole permet de faire une division ?

Question 3

Combien de degrés correspondent à une rotation complète ?

Question 4

Quelle formule permet de calculer le périmètre d'une roue ?

Question 5

Une roue a un diamètre de 56 mm. Pourquoi le robot avance t-il de plus de 56 mm lorsqu'elle effectue une rotation complète ?

Question 6

Pourquoi la distance réellement parcourue peut-elle être légèrement différente de la distance calculée ?

Correction

Réponse 1

```
*
```

Réponse 2

```
/
```

Réponse 3

```
360 degrés
```

Réponse 4

```
Périmètre = π × diamètre
```

Réponse 5

Parce qu'une rotation complète correspond au périmètre de la roue et non à son diamètre.

Réponse 6

À cause notamment du glissement, du sol, de la précision des roues, de la batterie et de la mécanique du robot.

Cours 4 : Les conditions `if`, `elif` et `else` avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

- comprendre ce qu'est une condition
- utiliser `if`
- utiliser `else`
- utiliser `elif`
- utiliser les opérateurs de comparaison
- faire prendre une décision à un programme
- faire réagir un robot EV3 à une information provenant d'un capteur

Matériel nécessaire

- 1 LEGO Education Mindstorms EV3
- 1 moteur EV3
- 1 capteur tactile EV3
- 1 ordinateur avec Visual Studio Code
- MicroPython installé sur l'EV3
- câble USB ou connexion permettant de transférer le programme

1. Rappel du cours précédent

Lors du cours précédent, nous avons appris à effectuer des calculs avec des variables.

Par exemple :

```
diametre = 56
distance = 500

perimetre = 3.14 * diametre
rotations = distance / perimetre
angle = rotations * 360
```

Jusqu'à présent, nos programmes exécutent simplement les instructions les unes après les autres.

Par exemple :

```
Avancer

Puis

Attendre

Puis

Reculer

Puis

S'arrêter
```

Mais un robot autonome doit être capable de prendre des décisions.

Par exemple :

```
SI un obstacle est détecté

ALORS s'arrêter
```

C'est exactement ce que permettent les conditions.

2. Qu'est-ce qu'une condition ?

Une condition permet au programme de vérifier si quelque chose est vrai ou faux.

Prenons un exemple simple :

```
SI la température est supérieure à 25°C

ALORS allumer le ventilateur
```

En Python, cela pourrait être écrit :

```
temperature = 30

if temperature > 25:
    print("Allumer le ventilateur")
```

Comme `30` est supérieur à `25`, le message est affiché.

3. La condition `if`

`if` signifie :

```
si
```

Sa structure est :

```
if condition:
    instruction
```

Exemple :

```
age = 20

if age >= 18:
    print("Majeur")
```

Le programme vérifie :

```
20 >= 18
```

La réponse est vraie.

Le programme affiche donc :

```
Majeur
```

4. Très important : l'indentation

En Python, les espaces placés au début des lignes sont importants.

Exemple correct :

```
age = 20

if age >= 18:
    print("Majeur")
```

La ligne :

```
print("Majeur")
```

est décalée vers la droite.

Cela signifie qu'elle appartient à la condition.

Exemple incorrect :

```
age = 20

if age >= 18:
print("Majeur")
```

Python produira une erreur.

À retenir

Après :

```
if condition:
```

les instructions qui appartiennent à la condition doivent être indentées.

5. Les opérateurs de comparaison

Pour créer des conditions, nous devons comparer des valeurs.

Supérieur à

```
>
```

Exemple :

```
vitesse = 500

if vitesse > 400:
    print("Vitesse élevée")
```

Inférieur à

```
<
```

Exemple :

```
distance = 10

if distance < 20:
    print("Obstacle proche")
```

Supérieur ou égal à

```
>=
```

Exemple :

```
age = 18

if age >= 18:
    print("Majeur")
```

Inférieur ou égal à

```
<=
```

Exemple :

```
batterie = 20

if batterie <= 20:
    print("Batterie faible")
```

Égal à

```
==
```

Exemple :

```
couleur = 5

if couleur == 5:
    print("Couleur détectée")
```

Attention à ne pas confondre :

```
=
```

et :

```
==
```

= sert à donner une valeur à une variable.

```
vitesse = 500
```

== sert à comparer deux valeurs.

```
if vitesse == 500:
```

Différent de

```
!=
```

Exemple :

```
vitesse = 300

if vitesse != 0:
    print("Le moteur est en mouvement")
```

6. Exercice 1

Que va afficher ce programme ?

```
distance = 15

if distance < 20:
    print("Obstacle")
```

Réponse :

```
Obstacle
```

Pourquoi ?

Parce que :

```
15 < 20
```

est vrai.

7. Exercice 2

Que va afficher ce programme ?

```
distance = 50

if distance < 20:
    print("Obstacle")
```

Réponse :

```
Rien
```

La condition est fausse.

Le programme ignore donc l'instruction située à l'intérieur du `if`.

8. La condition `else`

`else` signifie :

```
sinon
```

Exemple :

```
age = 15

if age >= 18:
    print("Majeur")
else:
    print("Mineur")
```

Le programme pose la question :

```
Est-ce que age est supérieur ou égal à 18 ?
```

Si oui :

```
Majeur
```

Sinon :

```
Mineur
```

9. Application robotique

Imaginons que nous connaissions la distance jusqu'à un obstacle.

```
distance = 15

if distance < 20:
    print("STOP")
else:
    print("AVANCER")
```

Le programme peut donc prendre une décision selon la situation.

C'est la base d'un robot autonome.

10. Plusieurs possibilités avec `elif`

`elif` permet de tester une nouvelle condition si la précédente est fausse.

Exemple :

```
distance = 35

if distance < 20:
    print("STOP")
elif distance < 50:
    print("RALENTIR")
else:
    print("AVANCER")
```

Le programme vérifie les conditions dans l'ordre.

Dans cet exemple :

```
distance = 35
```

Première question :

```
35 < 20 ?
```

Non.

Deuxième question :

```
35 < 50 ?
```

Oui.

Le programme affiche donc :

```
RALENTIR
```

11. Comprendre l'ordre des conditions

Prenons :

```
distance = 10

if distance < 20:
    print("STOP")
elif distance < 50:
    print("RALENTIR")
else:
    print("AVANCER")
```

`10` est inférieur à `20`.

La première condition est donc vraie.

Python exécute :

```
STOP
```

et ne teste pas les conditions suivantes.

L'ordre des conditions est donc important.

12. Exercice 3

Complète ce programme.

Si la vitesse dépasse `700`, afficher :

```
Trop rapide
```

Sinon afficher :

```
Vitesse correcte
```

Solution :

```
vitesse = 800

if vitesse > 700:
    print("Trop rapide")
else:
    print("Vitesse correcte")
```

13. Premier capteur : le capteur tactile

Le capteur tactile peut détecter s'il est appuyé.



Il possède principalement deux états :

```
Appuyé

Non appuyé
```

Nous pouvons donc le comparer à une sorte d'interrupteur.

14. Initialiser le capteur tactile

Connectons le capteur sur le port 1.

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port

capteur = TouchSensor(Port.S1)
```

Nous pouvons ensuite vérifier son état :

```
if capteur.pressed():
    print("Bouton appuyé")
```

15. Programme complet

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port

ev3 = EV3Brick()

capteur = TouchSensor(Port.S1)

if capteur.pressed():
    ev3.speaker.beep()
```

Le robot produit un son uniquement si le capteur est appuyé au moment où la condition est vérifiée.

16. Exercice pratique

Modifier le programme pour afficher :

```
Bouton appuyé
```

si le capteur est appuyé.

Sinon afficher :

```
Bouton libre
```

Solution :

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port

capteur = TouchSensor(Port.S1)

if capteur.pressed():
    print("Bouton appuyé")
else:
    print("Bouton libre")
```

17. Faire réagir un moteur

Connectons :

```
Moteur sur le port A

Capteur tactile sur le port 1
```

Programme :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

if capteur.pressed():
    moteur.stop()
else:
    moteur.run(500)
```

Si le capteur est appuyé :

```
le moteur s'arrête
```

Sinon :

```
le moteur tourne
```

18. Problème important

Testons le programme précédent.

Que se passe t-il ?

Le programme vérifie le capteur une seule fois.

Puis il se termine.

Cela signifie que le robot ne surveille pas continuellement le capteur.

Pour résoudre ce problème, nous aurons besoin d'une notion que nous verrons au prochain cours :

`les boucles`

19. Conditions avec plusieurs nombres

Créons un système de vitesse.

```
distance = 70

if distance < 20:
    vitesse = 0
elif distance < 50:
    vitesse = 200
else:
    vitesse = 500

print(vitesse)
```

Avec :

```
distance = 70
```

le résultat est :

```
500
```

20. Exercice 4

Complète les valeurs obtenues.

Programme :

```
distance = 10

if distance < 20:
    vitesse = 0
elif distance < 50:
    vitesse = 200
else:
    vitesse = 500
```

Quelle est la valeur de `vitesse` ?

Réponse :

```
0
```

Avec :

```
distance = 30
```

Réponse :

```
200
```

Avec :

```
distance = 80
```

Réponse :

```
500
```

21. Les booléens

Nous avons vu lors du cours précédent le type :

```
bool
```

Un booléen peut contenir :

```
True
```

ou :

```
False
```

Exemple :

```
obstacle = True

if obstacle:
    print("STOP")
```

Cela revient à dire :

```
SI obstacle est vrai

ALORS afficher STOP
```

22. Exemple EV3

La fonction :

```
capteur.pressed()
```

renvoie justement une valeur de type booléen.

Elle renvoie :

```
True
```

si le capteur est appuyé.

Et :

```
False
```

s'il ne l'est pas.

Nous pourrions écrire :

```
etat = capteur.pressed()

print(etat)
```

Puis appuyer ou non sur le capteur avant d'exécuter le programme.

23. Combiner plusieurs conditions

Python permet également de vérifier plusieurs choses.

Nous utiliserons principalement :

```
and
```

et :

```
or
```

24. Utiliser `and`

`and` signifie :

```
et
```

Exemple :

```
distance = 30
vitesse = 500

if distance < 50 and vitesse > 400:
    print("Attention")
```

Pour que le message soit affiché, les deux conditions doivent être vraies.

```
distance < 50

ET

vitesse > 400
```

25. Utiliser `or`

`or` signifie :

```
ou
```

Exemple :

```
batterie = 10
distance = 5

if batterie < 20 or distance < 10:
    print("Arrêt de sécurité")
```

Une seule des deux conditions doit être vraie pour déclencher l'action.

26. Exercice 5

Que va afficher :

```
distance = 15
vitesse = 600

if distance < 20 and vitesse > 500:
    print("Danger")
```

Réponse :

```
Danger
```

Les deux conditions sont vraies.

27. Le mot clé `not`

`not` permet d'inverser une condition.

Exemple :

```
bouton = False

if not bouton:
    print("Bouton non appuyé")
```

Avec le capteur EV3 :

```
if not capteur.pressed():
    print("Capteur libre")
```

28. Mini défi 1

Créer un programme qui vérifie une variable :

```
batterie = 30
```

Le programme doit afficher :

```
Batterie faible
```

si la valeur est inférieure ou égale à `20`.

Sinon :

```
Batterie correcte
```

Solution :

```
batterie = 30

if batterie <= 20:
    print("Batterie faible")
else:
    print("Batterie correcte")
```

29. Mini défi 2

Créer trois niveaux de vitesse.

Si :

```
distance < 20
```

alors :

```
vitesse = 0
```

Si :

```
distance < 50
```

alors :

```
vitesse = 200
```

Sinon :

```
vitesse = 500
```

Solution :

```
distance = 35

if distance < 20:
    vitesse = 0
elif distance < 50:
    vitesse = 200
else:
    vitesse = 500

print(vitesse)
```

30. Mini défi 3

Connecter :

```
Moteur A

Capteur tactile S1
```

Le programme doit :

```
faire tourner le moteur
```

si le bouton n'est pas appuyé.

Et :

```
arrêter le moteur
```

si le bouton est appuyé.

Solution :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

if capteur.pressed():
    moteur.stop()
else:
    moteur.run(500)
```

31. Défi final : système de sécurité

Nous allons imaginer qu'un moteur est utilisé dans une machine.

Nous possédons :

```
un bouton de sécurité

une vitesse
```

Le moteur ne peut fonctionner que si :

```
le bouton n'est pas appuyé
```

et :

```
la vitesse ne dépasse pas 700
```

Programme de départ :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

vitesse = 500
```

Compléter le programme.

Solution possible

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

vitesse = 500

if not capteur.pressed() and vitesse <= 700:
    moteur.run(vitesse)
else:
    moteur.stop()
```

32. Comprendre le problème du programme

Même avec cette solution, le programme ne vérifie le capteur qu'une seule fois.

Si le moteur commence à tourner puis que nous appuyons sur le bouton après le test, le programme ne reviendra pas automatiquement vérifier le capteur.

Nous avons donc besoin de quelque chose comme :

```
Vérifier le capteur
  ↓
Prendre une décision
  ↓
Recommencer
  ↓
Vérifier le capteur
  ↓
Prendre une décision
  ↓
Recommencer
```

C'est exactement le rôle d'une boucle.

33. Questions de fin de cours

Question 1

À quoi sert `if` ?

Question 2

Quelle est la différence entre :

`=`

et :

`==`

Question 3

Que signifie :

`!=`

Question 4

À quoi sert `else` ?

Question 5

À quoi sert `elif` ?

Question 6

Que signifie :

`and`

Question 7

Que signifie :

`or`

Question 8

Que retourne :

```
capteur.pressed()
```

Correction

Réponse 1

`if` permet d'exécuter une action seulement si une condition est vraie.

Réponse 2

= permet d'attribuer une valeur.

```
vitesse = 500
```

== permet de comparer deux valeurs.

```
vitesse == 500
```

Réponse 3

`!=` signifie :

```
différent de
```

Réponse 4

`else` indique ce qui doit être exécuté si la condition précédente est fausse.

Réponse 5

`elif` permet de tester une autre condition.

Réponse 6

`and` signifie que plusieurs conditions doivent être vraies en même temps.

Réponse 7

`or` signifie qu'au moins une des conditions doit être vraie.

Réponse 8

La fonction retourne :

```
True
```

ou :

```
False
```

selon que le capteur est appuyé ou non.

Cours 5 : Les boucles `while` avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre ce qu'est une boucle
2. comprendre le fonctionnement de `while`
3. créer une boucle infinie
4. créer une boucle avec une condition
5. utiliser un capteur dans une boucle
6. faire réagir continuellement un robot EV3
7. comprendre le risque d'une boucle infinie
8. utiliser `break` pour arrêter une boucle

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 1 moteur EV3
3. 1 capteur tactile EV3
4. 1 ordinateur avec Visual Studio Code
5. MicroPython installé sur l'EV3
6. câble USB ou connexion permettant de transférer le programme

1. Rappel du cours précédent

Lors du cours précédent, nous avons appris à utiliser les conditions.

Par exemple :

```
distance = 15

if distance < 20:
    print("STOP")
else:
    print("AVANCER")
```

Le programme vérifie la valeur de `distance`.

Si la distance est inférieure à `20`, il affiche :

```
STOP
```

Sinon, il affiche :

```
AVANCER
```

Nous avons également utilisé un capteur tactile.

```
if capteur.pressed():
    moteur.stop()
else:
    moteur.run(500)
```

Mais ce programme possède un problème important.

La condition n'est vérifiée qu'une seule fois.

2. Le problème

Imaginons que le capteur ne soit pas appuyé au lancement du programme.

Le robot exécute :

```
moteur.run(500)
```

Il commence donc à avancer.

Ensuite, le programme arrive à la fin.

Si nous appuyons sur le capteur quelques secondes plus tard, le programme ne vérifie plus son état.

Nous aimerions plutôt obtenir ce comportement :

```
Vérifier le capteur

Puis

prendre une décision

Puis

vérifier à nouveau

Puis

prendre une décision

Puis

recommencer
```

Pour cela, nous allons utiliser une boucle.

3. Qu'est-ce qu'une boucle ?

Une boucle permet de répéter une ou plusieurs instructions.

Imaginons que nous voulions afficher cinq fois :

```
Bonjour
```

Nous pourrions écrire :

```
print("Bonjour")
print("Bonjour")
print("Bonjour")
print("Bonjour")
print("Bonjour")
```

Cela fonctionne.

Mais ce n'est pas très pratique.

Imagine maintenant que nous voulions afficher le message 1000 fois.

Il serait beaucoup plus intéressant de demander à Python de répéter automatiquement l'instruction.

4. La boucle `while`

`while` signifie :

```
tant que
```

La structure générale est :

```
while condition:
    instruction
```

Cela signifie :

```
TANT QUE la condition est vraie

répéter l'instruction
```

5. Premier exemple

```
nombre = 0

while nombre < 5:
    print(nombre)
    nombre = nombre + 1
```

Analysons ce programme.

Au départ :

```
nombre = 0
```

Python vérifie :

```
0 < 5
```

C'est vrai.

Il affiche donc :

```
0
```

Puis :

```
nombre = nombre + 1
```

La valeur devient :

```
1
```

Python revient ensuite au début de la boucle.

Il vérifie :

```
1 < 5
```

C'est toujours vrai.

Le processus continue.

Le programme affiche finalement :

```
0
1
2
3
4
```

Lorsque `nombre` devient `5`, la condition :

```
5 < 5
```

est fausse.

La boucle s'arrête.

6. Visualiser le fonctionnement

On peut représenter la boucle ainsi :

```
nombre = 0

↓

nombre < 5 ?

↓

OUI

↓

Afficher nombre

↓

Ajouter 1

↓

Retour au test
```

Lorsque la réponse devient :

```
NON
```

la boucle se termine.

7. Exercice 1

Que va afficher ce programme ?

```
compteur = 1

while compteur <= 4:
    print(compteur)
    compteur = compteur + 1
```

Réponse :

```
1
2
3
4
```

8. Exercice 2

Complète le programme pour afficher les nombres de `0` à `9`.

```
nombre = 0

while __________:
    print(nombre)
    nombre = nombre + 1
```

Solution :

```
nombre = 0

while nombre < 10:
    print(nombre)
    nombre = nombre + 1
```

9. Attention aux boucles infinies

Regardons ce programme :

```
nombre = 0

while nombre < 5:
    print(nombre)
```

Quel est le problème ?

La variable `nombre` ne change jamais.

Elle reste toujours égale à :

```
0
```

La condition :

```
0 < 5
```

reste donc toujours vraie.

Le programme affiche continuellement :

```
0
0
0
0
0
0
0
...
```

On appelle cela une boucle infinie.

10. Une boucle infinie peut être volontaire

En robotique, une boucle infinie est souvent très utile.

Un robot autonome doit généralement surveiller son environnement en permanence.

Nous pouvons écrire :

```
while True:
    print("Robot actif")
```

`True` signifie :

```
vrai
```

La condition est donc toujours vraie.

Le programme recommence continuellement.

11. Premier exemple EV3

Nous allons faire produire continuellement un son à la brique.

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

while True:
    ev3.speaker.beep()
    wait(1000)
```

Le programme effectue :

```
BIP

attendre 1 seconde

BIP

attendre 1 seconde

BIP

attendre 1 seconde
```

et continue tant que le programme n'est pas arrêté.

12. Pourquoi ajouter `wait()` ?

Essayons :

```
while True:
    print("Bonjour")
```

Le programme va essayer d'afficher le message aussi rapidement que possible.

Cela peut provoquer énormément d'exécutions en très peu de temps.

Dans certains programmes robotiques, il peut donc être utile d'ajouter une petite attente.

Par exemple :

```
while True:
    print("Bonjour")
    wait(100)
```

Ici, le programme attend :

```
100 millisecondes
```

entre chaque répétition.

13. Capteur tactile et boucle infinie

Nous pouvons maintenant résoudre le problème du cours précédent.

Connectons :

```
Capteur tactile sur le port 1
```

Programme :

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():
        print("Bouton appuyé")
    else:
        print("Bouton libre")

    wait(100)
```

Cette fois, le capteur est vérifié continuellement.

14. Observer le comportement

Lorsque le bouton n'est pas appuyé :

```
Bouton libre
```

Lorsque nous appuyons :

```
Bouton appuyé
```

Lorsque nous relâchons :

```
Bouton libre
```

Le robot surveille donc continuellement le capteur.

15. Ajouter un moteur

Connectons maintenant :

```
Moteur sur le port A

Capteur tactile sur le port 1
```

Programme :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():
        moteur.stop()
    else:
        moteur.run(500)
    wait(10)
```

16. Comprendre le programme

Le robot effectue continuellement :

```
Lire le capteur
  ↓
Le bouton est appuyé ?
  ↓
OUI
Arrêter le moteur
  ↓
NON
Faire tourner le moteur
  ↓
Recommencer
```

Nous avons maintenant un système qui réagit en temps réel.

17. Exercice pratique 1

Modifier le programme pour obtenir :

```
Bouton libre
  ↓
Moteur à 300 degrés par seconde
```

et :

```
Bouton appuyé
  ↓
Moteur à 800 degrés par seconde
```

Une solution possible :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():
        moteur.run(800)
    else:
        moteur.run(300)

    wait(10)
```

18. Boucle avec une condition

Une boucle ne doit pas forcément être infinie.

Nous pouvons écrire :

```
nombre = 0

while nombre < 5:
    print(nombre)
    nombre = nombre + 1
```

La boucle continue uniquement :

```
tant que nombre est inférieur à 5
```

Avec un capteur, nous pouvons faire la même chose.

19. Avancer tant que le bouton n’est pas appuyé

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while not capteur.pressed():
    moteur.run(500)

moteur.stop()
```

Cette instruction :

```
while not capteur.pressed():
```

signifie :

```
TANT QUE le capteur n’est pas appuyé
```

le moteur continue de tourner.

Lorsque le capteur est appuyé, la condition devient fausse.

Le programme sort de la boucle.

Puis il exécute :

```
moteur.stop()
```

20. Comparaison entre deux approches

Première approche :

```
while True:

    if capteur.pressed():
        moteur.stop()
    else:
        moteur.run(500)
```

Cette boucle continue indéfiniment.

Le robot peut recommencer à avancer lorsque le bouton est relâché.

Deuxième approche :

```
while not capteur.pressed():
    moteur.run(500)

moteur.stop()
```

Cette boucle se termine lorsque le bouton est appuyé.

Le robot ne recommence pas automatiquement à avancer.

Ces deux comportements sont différents.

21. Expérience

Teste les deux programmes.

Programme A

```
while True:

    if capteur.pressed():
        moteur.stop()
    else:
        moteur.run(500)
```

Appuie sur le bouton.

Relâche le bouton.

Que se passe t-il ?

Programme B

```
while not capteur.pressed():
    moteur.run(500)

moteur.stop()
```

Appuye sur le bouton.

Relâche le bouton.

Que se passe t-il ?

L'objectif est que tu remarques la différence entre :

`surveiller continuellement`

et :

`attendre qu'un événement arrive`

22. Utiliser `break`

Python permet également de sortir manuellement d'une boucle.

Pour cela, nous pouvons utiliser :

`break`

Exemple :

```
nombre = 0

while True:

    print(nombre)

    nombre = nombre + 1

    if nombre == 5:
        break
```

Lorsque `nombre` devient `5`, Python exécute :

`break`

La boucle s'arrête.

23. `break` avec l'EV3

Nous pouvons écrire :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while True:

    moteur.run(500)

    if capteur.pressed():
        break

moteur.stop()
```

Le moteur tourne tant que le capteur n'est pas appuyé.

Lorsque nous appuyons :

```
break
```

arrête la boucle.

Puis :

```
moteur.stop()
```

est exécuté.

24. Quelle méthode choisir ?

Ces deux programmes ont un comportement similaire.

Méthode 1

```
while not capteur.pressed():
    moteur.run(500)

moteur.stop()
```

Méthode 2

```
while True:

    moteur.run(500)

    if capteur.pressed():
        break

moteur.stop()
```

La première solution est plus simple pour ce cas.

La deuxième devient intéressante lorsque plusieurs événements peuvent provoquer la sortie de la boucle.

25. Utiliser un compteur

Une boucle peut également compter le nombre d'actions effectuées.

```
compteur = 0

while compteur < 3:

    print("BIP")

    compteur = compteur + 1
```

Résultat :

```
BIP
BIP
BIP
```

26. Application EV3

Faire produire trois sons à la brique :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

compteur = 0

while compteur < 3:

    ev3.speaker.beep()

    wait(500)

    compteur = compteur + 1
```

27. Exercice pratique 2

Modifier le programme pour produire :

```
5 sons
```

avec :

```
1 seconde
```

entre chaque son.

Solution :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

compteur = 0

while compteur < 5:

    ev3.speaker.beep()

    wait(1000)

    compteur = compteur + 1
```

28. Faire varier la vitesse dans une boucle

Nous pouvons utiliser une boucle pour modifier progressivement une variable.

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 100

while vitesse <= 800:

    moteur.run(vitesse)

    wait(1000)

    vitesse = vitesse + 100

moteur.stop()
```

Le moteur passe successivement par :

```
100

200

300

400

500

600

700

800
```

degrés par seconde.

29. Exercice pratique 3

Créer un programme qui commence avec :

```
vitesse = 200
```

et augmente la vitesse de :

```
200 degrés par seconde
```

toutes les secondes.

La boucle doit s'arrêter lorsque la vitesse dépasse :

```
1000 degrés par seconde
```

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 200

while vitesse <= 1000:

    moteur.run(vitesse)

    wait(1000)

    vitesse = vitesse + 200

moteur.stop()
```

30. Combiner `while` et `if`

Voici une structure que nous utiliserons énormément en robotique :

```
while True:

    if condition:
        action
    else:
        autre_action
```

Exemple :

```
while True:

    if capteur.pressed():
        moteur.stop()
    else:
        moteur.run(500)
```

Cette combinaison permet au robot de :

```
observer

↓

prendre une décision

↓

agir

↓

observer à nouveau
```

31. Premier véritable robot anticollision

Construisons maintenant un robot avec un capteur tactile placé à l'avant.

Lorsqu'il touche un obstacle, il doit :

```
s'arrêter

↓

reculer pendant 1 seconde

↓

s'arrêter
```

Programme :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():

        moteur.stop()

        moteur.run(-500)
        wait(1000)

        moteur.stop()

        break

    else:

        moteur.run(500)
```

32. Comprendre le comportement

Au démarrage :

```
Le moteur tourne
```

Puis le programme vérifie continuellement le capteur.

Lorsque le robot touche quelque chose :

```
Capteur appuyé

↓

Arrêt

↓

Marche arrière

↓

Attendre 1 seconde

↓

Arrêt

↓

Fin du programme
```

33. Améliorer le robot anticollision

Nous pouvons maintenant supprimer `break`.

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():

        moteur.stop()

        moteur.run(-500)
        wait(1000)

    else:

        moteur.run(500)
```

Cette fois, le programme continue après avoir rencontré un obstacle.

Après avoir reculé, le robot recommence à vérifier le capteur.

34. Un problème apparaît

Avec un seul moteur, notre robot peut simplement :

```
avancer

ou

reculer
```

Pour réellement contourner un obstacle, nous aurons besoin de deux moteurs.

Avec deux moteurs, nous pourrons :

```
avancer

reculer

tourner à gauche

tourner à droite
```

Nous aborderons cela prochainement.

35. Mini défi 1

Créer un programme qui :

```
fait tourner le moteur

↓

attend que le bouton soit appuyé

↓

arrête le moteur
```

Solution :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while not capteur.pressed():
    moteur.run(500)

moteur.stop()
```

36. Mini défi 2

Créer un programme qui :

```
fait tourner le moteur à 300 degrés par seconde

↓

si le bouton est appuyé

↓

fait tourner le moteur à 800 degrés par seconde
```

Le programme doit continuer indéfiniment.

Solution :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():
        moteur.run(800)
    else:
        moteur.run(300)

    wait(10)
```

37. Mini défi 3

Créer un compteur.

Le programme doit afficher :

```
1
2
3
4
5
6
7
8
9
10
```

en utilisant une boucle `while`.

Solution :

```
nombre = 1

while nombre <= 10:

    print(nombre)

    nombre = nombre + 1
```

38. Défi final : le moteur intelligent

Créer un programme avec :

```
1 moteur

1 capteur tactile
```

Le comportement doit être :

```
Le moteur tourne à 400 degrés par seconde

↓

Le bouton est appuyé

↓

Le moteur s'arrête

↓

La brique fait un son

↓

Le moteur tourne dans le sens inverse pendant 2 secondes

↓

Le moteur s'arrête

↓

Le programme se termine
```

Solution possible

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

ev3 = EV3Brick()

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

vitesse = 400

while not capteur.pressed():
    moteur.run(vitesse)

moteur.stop()

ev3.speaker.beep()

moteur.run(-vitesse)

wait(2000)

moteur.stop()
```

39. Défi supplémentaire

Modifier le programme précédent pour que le robot recommence automatiquement après avoir reculé.

Le fonctionnement doit devenir :

```
Avancer
↓
Toucher un obstacle
↓
Faire un son
↓
Reculer
↓
Avancer
↓
Toucher un nouvel obstacle
↓
Faire un son
↓
Reculer
↓
Continuer
```

Une solution possible

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

ev3 = EV3Brick()

moteur = Motor(Port.A)
capteur = TouchSensor(Port.S1)

vitesse = 400

while True:

    if capteur.pressed():

        moteur.stop()

        ev3.speaker.beep()

        moteur.run(-vitesse)

        wait(2000)

    else:

        moteur.run(vitesse)

    wait(10)
```

40. Questions de fin de cours

Question 1

À quoi sert une boucle ?

Question 2

Que signifie `while` ?

Question 3

Que signifie :

```
while True:
```

Question 4

Pourquoi ce programme ne se termine t il jamais ?

```
nombre = 0

while nombre < 10:
    print(nombre)
```

Question 5

À quoi sert :

```
break
```

Question 6

Quelle différence existe entre :

```
while True:
```

et :

```
while not capteur.pressed():
```

Question 7

Pourquoi les boucles sont-elles particulièrement importantes en robotique ?

Correction

Réponse 1

Une boucle permet de répéter une ou plusieurs instructions.

Réponse 2

`while` signifie :

`tant que`

Réponse 3

Cela crée une boucle dont la condition est toujours vraie.

Elle continue donc jusqu'à ce que le programme soit arrêté ou qu'une instruction comme `break` permette d'en sortir.

Réponse 4

Parce que `nombre` reste toujours égal à `0`.

Il manque une instruction comme :

```
nombre = nombre + 1
```

Réponse 5

`break` permet de sortir immédiatement d'une boucle.

Réponse 6

```
while True:
```

continue indéfiniment.

Alors que :

```
while not capteur.pressed():
```

continue seulement tant que le capteur n'est pas appuyé.

Réponse 7

Parce qu'un robot doit généralement observer continuellement ses capteurs pour pouvoir réagir aux changements de son environnement.

Erreurs fréquentes

Erreur 1 : oublier les deux points

Incorrect :

```
while nombre < 10
```

Correct :

```
while nombre < 10:
```

Erreur 2 : mauvaise indentation

Incorrect :

```
while nombre < 10:
print(nombre)
```

Correct :

```
while nombre < 10:
    print(nombre)
```

Erreur 3 : oublier de modifier la variable

Programme problématique :

```
nombre = 0
while nombre < 10:
    print(nombre)
```

La boucle ne se termine jamais.

Programme correct :

```
nombre = 0
while nombre < 10:
    print(nombre)
    nombre = nombre + 1
```

Erreur 4 : oublier d'arrêter le moteur

Tu pourrais écrire :

```
while not capteur.pressed():
    moteur.run(500)
```

mais oublier :

```
moteur.stop()
```

après la boucle.
Il faut bien comprendre que sortir de la boucle n'implique pas automatiquement l'arrêt du moteur.

Cours 6 : Les boucles `for` avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le fonctionnement d'une boucle `for`
2. utiliser `range()`
3. répéter une action un nombre précis de fois
4. utiliser une variable de boucle
5. comparer `for` et `while`
6. utiliser une boucle `for` avec un moteur EV3
7. créer des séquences répétitives
8. préparer des déplacements répétitifs pour le robot

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 1 moteur EV3
3. 1 ordinateur avec Visual Studio Code
4. MicroPython installé sur l'EV3
5. câble USB ou connexion permettant de transférer le programme

1. Rappel du cours précédent

Lors du cours précédent, nous avons découvert la boucle `while`.

Par exemple :

```
nombre = 0

while nombre < 5:
    print(nombre)
    nombre = nombre + 1
```

Cette boucle affiche :

```
0
1
2
3
4
```

Nous avons également utilisé :

```
while True:
```

pour créer un programme qui se répète continuellement.

Mais si nous savons déjà exactement combien de fois nous voulons répéter une action, Python propose une solution souvent plus simple.

La boucle `for`.

2. Le problème des répétitions

Imaginons que nous voulions faire produire cinq sons à la brique EV3.

Nous pourrions écrire :

```
ev3.speaker.beep()
wait(500)

ev3.speaker.beep()
wait(500)

ev3.speaker.beep()
wait(500)

ev3.speaker.beep()
wait(500)

ev3.speaker.beep()
wait(500)
```

Ce programme fonctionne.

Mais il contient beaucoup de répétitions.

Nous pouvons faire plus simple.

```
for i in range(5):
    ev3.speaker.beep()
    wait(500)
```

Cette boucle demande à Python de répéter les instructions cinq fois.

3. La boucle `for`

La structure générale est :

```
for variable in range(nombre):
    instruction
```

Exemple :

```
for i in range(5):
    print("Bonjour")
```

Le programme affiche :

```
Bonjour
Bonjour
Bonjour
Bonjour
Bonjour
```

4. À quoi sert `range()` ?

`range()` permet de produire une suite de nombres.

Par exemple :

```
range(5)
```

correspond aux nombres :

```
0
1
2
3
4
```

Il y a donc bien cinq valeurs.

C'est pour cette raison que :

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

répète la boucle cinq fois.

5. Observer la variable de boucle

Essayons :

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

Le programme affiche :

```
0
1
2
3
4
```

La variable `i` change automatiquement à chaque répétition.

Première répétition :

```
i = 0
```

Deuxième répétition :

```
i = 1
```

Troisième répétition :

```
i = 2
```

et ainsi de suite.

6. Pourquoi commencer à zéro ?

En programmation, il est très courant de commencer à compter à partir de zéro.

Avec :

```
range(5)
```

Python génère cinq nombres :

```
0
1
2
3
4
```

Il ne génère pas :

```
1
2
3
4
5
```

C’est un détail important à retenir.

7. Exercice 1

Que va afficher ce programme ?

```
for i in range(3):
    print(i)
```

Réponse :

```
0
1
2
```

8. Exercice 2

Combien de fois le mot `Robot` sera t il affiché ?

```
for i in range(10):
    print("Robot")
```

Réponse :

```
10 fois
```

9. Premier programme EV3 avec `for`

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

for i in range(5):
    ev3.speaker.beep()
    wait(500)
```

La brique produit cinq sons.

10. Exercice pratique 1

Modifier le programme pour produire :

```
10 sons
```

avec :

```
250 millisecondes
```

entre chaque son.

Solution :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

for i in range(10):
    ev3.speaker.beep()
    wait(250)
```

11. Comparaison avec `while`

Avec `while` :

```
compteur = 0

while compteur < 5:
    print("Bonjour")
    compteur = compteur + 1
```

Avec `for` :

```
for i in range(5):
    print("Bonjour")
```

La boucle `for` est généralement plus simple lorsque nous connaissons le nombre de répétitions.

12. Quand utiliser `while` ?

`while` est particulièrement utile lorsque nous ne savons pas combien de temps la boucle doit continuer.

Par exemple :

```
while not capteur.pressed():
    moteur.run(500)
```

Nous ne savons pas quand l'utilisateur appuiera sur le capteur.

La boucle continue donc tant que la condition reste vraie.

13. Quand utiliser `for` ?

`for` est particulièrement utile lorsque nous savons combien de fois nous voulons répéter une action.

Par exemple :

```
for i in range(4):
    ev3.speaker.beep()
```

Nous savons que l'action doit être répétée exactement quatre fois.

14. Utiliser `range()` avec un début et une fin

Nous pouvons également écrire :

```
for i in range(1, 6):
    print(i)
```

Résultat :

```
1
2
3
4
5
```

Le premier nombre est inclus.

Le deuxième indique la limite qui n'est pas incluse.

Ainsi :

```
range(1, 6)
```

produit :

```
1
2
3
4
5
```

15. Exercice 3

Que va afficher :

```
for i in range(3, 8):
    print(i)
```

Réponse :

```
3
4
5
6
7
```

16. Utiliser un pas

`range()` peut également recevoir une troisième valeur.

```
range(debut, fin, pas)
```

Exemple :

```
for i in range(0, 11, 2):
    print(i)
```

Résultat :

```
0
2
4
6
8
10
```

La valeur augmente de `2` à chaque répétition.

17. Compter à rebours

Il est également possible d'utiliser un pas négatif.

```
for i in range(5, 0, -1):
    print(i)
```

Résultat :

```
5
4
3
2
1
```

Nous pouvons ensuite afficher :

```
print("GO")
```

Programme complet :

```
for i in range(5, 0, -1):
    print(i)

print("GO")
```

18. Exercice pratique 2

Créer un compte à rebours de :

```
10
```

jusqu'à :

```
1
```

Puis afficher :

```
DÉPART
```

Solution :

```
for i in range(10, 0, -1):
    print(i)

print("DÉPART")
```

19. Faire varier la vitesse d'un moteur

Nous pouvons utiliser la variable `i` pour contrôler un moteur.

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for vitesse in range(100, 901, 100):
    moteur.run(vitesse)
    wait(500)

moteur.stop()
```

Les vitesses seront :

```
100
200
300
400
500
600
700
800
900
```

degrés par seconde.

20. Comprendre le programme

Dans :

```
for vitesse in range(100, 901, 100):
```

la variable `vitesse` prend successivement les valeurs :

```
100
200
300
400
500
600
700
800
900
```

Nous pouvons donc directement écrire :

```
moteur.run(vitesse)
```

Cela permet de faire accélérer progressivement le moteur.

21. Expérience

Teste :

```
for vitesse in range(100, 1001, 100):
    moteur.run(vitesse)
    wait(500)

moteur.stop()
```

Observer comment la vitesse augmente.

Puis tester :

```
for vitesse in range(1000, 99, -100):
    moteur.run(vitesse)
    wait(500)

moteur.stop()
```

Cette fois, le moteur ralentit progressivement.

22. Faire tourner un moteur plusieurs fois

Nous avons déjà utilisé :

```
moteur.run_angle(500, 360)
```

pour demander une rotation complète.

Nous pouvons maintenant répéter cette action.

```
for i in range(4):
    moteur.run_angle(500, 360)
```

Le moteur effectue :

```
4 rotations complètes
```

23. Exercice pratique 3

Faire effectuer au moteur :

```
5 rotations
```

avec une vitesse de :

```
400 degrés par seconde
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

for i in range(5):
    moteur.run_angle(400, 360)
```

24. Ajouter une pause entre les rotations

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for i in range(5):
    moteur.run_angle(400, 360)
    wait(1000)
```

Le moteur effectue une rotation.

Puis attend une seconde.

Puis recommence.

25. Alterner les directions

Nous pouvons utiliser une boucle pour faire tourner le moteur dans un sens puis dans l'autre.

```
for i in range(3):

    moteur.run_angle(500, 360)

    moteur.run_angle(500, -360)
```

Le moteur effectue :

```
1 rotation vers l'avant

1 rotation vers l'arrière
```

trois fois.

26. Défi pratique 1

Créer un programme qui effectue quatre fois :

```
1 rotation vers l'avant

attendre 500 ms

1 rotation vers l'arrière

attendre 500 ms
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for i in range(4):

    moteur.run_angle(500, 360)
    wait(500)

    moteur.run_angle(500, -360)
    wait(500)
```

27. Utiliser le numéro de répétition

Nous pouvons afficher le numéro de chaque répétition.

```
for i in range(5):
    print("Tour numéro")
    print(i)
```

Mais le premier tour sera :

```
0
```

Si nous voulons afficher :

```
1
2
3
4
5
```

nous pouvons utiliser :

```
for i in range(5):
    print(i + 1)
```

28. Application EV3

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

for i in range(5):

    print("Répétition")
    print(i + 1)

    ev3.speaker.beep()

    wait(500)
```

29. Calculer avec la variable de boucle

La variable de boucle peut être utilisée dans des calculs.

```
for i in range(5):
    vitesse = 200 + i * 100
    print(vitesse)
```

Résultat :

```
200
300
400
500
600
```

30. Application moteur

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for i in range(5):

    vitesse = 200 + i * 100

    moteur.run(vitesse)

    wait(1000)

moteur.stop()
```

Le moteur accélère automatiquement à chaque répétition.

31. Exercice pratique 4

Créer un programme qui réalise les vitesses suivantes :

```
300
500
700
900
```

Utiliser une boucle `for`.

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for vitesse in range(300, 901, 200):

    moteur.run(vitesse)

    wait(1000)

moteur.stop()
```

32. Boucles imbriquées

Une boucle peut contenir une autre boucle.

Exemple :

```
for groupe in range(3):

    for i in range(2):
        print("BIP")

    print("Pause")
```

Le résultat sera :

```
BIP
BIP
Pause
BIP
BIP
Pause
BIP
BIP
Pause
```

On appelle cela des boucles imbriquées.

33. Application EV3

Faire trois séries de deux sons.

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

for groupe in range(3):

    for i in range(2):
        ev3.speaker.beep()
        wait(250)

    wait(1000)
```

34. Attention à l'indentation

Dans :

```
for groupe in range(3):

    for i in range(2):
        ev3.speaker.beep()
        wait(250)

    wait(1000)
```

`wait(250)` appartient à la deuxième boucle.

Mais :

```
wait(1000)
```

appartient seulement à la première boucle.

L'indentation permet donc de comprendre la structure du programme.

35. Préparation aux déplacements du robot

Imaginons maintenant un robot équipé de deux moteurs.

Pour faire un carré, nous aurons besoin de répéter quatre fois :

```
Avancer

Puis

Tourner de 90 degrés
```

Sans boucle :

```
Avancer
Tourner

Avancer
Tourner

Avancer
Tourner

Avancer
Tourner
```

Avec une boucle :

```
for i in range(4):
    avancer
    tourner
```

Nous verrons prochainement comment remplacer `avancer` et `tourner` par de véritables commandes EV3.

36. Exercice de réflexion

Un robot doit faire un triangle.

Combien de fois devra t il répéter :

```
Avancer

Puis

Tourner
```

Réponse :

```
3 fois
```

Nous pourrions donc imaginer :

```
for i in range(3):
    avancer
    tourner
```

37. Le carré

Un carré possède :

```
4 côtés
```

Nous pouvons donc préparer :

```
for i in range(4):
    moteur.run_angle(500, 1000)
```

Avec un seul moteur, cela ne crée évidemment pas encore un véritable carré.

Mais lorsque nous utiliserons deux moteurs, cette structure deviendra extrêmement utile.

38. `for` ou `while` ?

Choisir la bonne boucle est important.

Situation 1

Faire produire cinq sons.

Choix recommandé :

`for`

Nous connaissons le nombre de répétitions.

Situation 2

Avancer jusqu'à toucher un obstacle.

Choix recommandé :

`while`

Nous ne savons pas combien de temps cela prendra.

Situation 3

Faire quatre côtés d'un carré.

Choix recommandé :

`for`

Nous savons qu'il y a quatre côtés.

Situation 4

Surveiller continuellement un capteur.

Choix recommandé :

`while`

Le programme doit fonctionner sans connaître à l'avance le nombre de répétitions.

39. Mini défi 2

Associe chaque situation avec `for` ou `while`.

A

Faire dix rotations.

Réponse :

`for`

B

Avancer jusqu’à ce qu’un bouton soit appuyé.

Réponse :

`while`

C

Faire cinq sons.

Réponse :

`for`

D

Surveiller un capteur indéfiniment.

Réponse :

`while`

40. Défi final : séquence moteur

Créer un programme qui réalise la séquence suivante trois fois :

```
Faire une rotation vers l’avant
↓
Attendre 500 ms
↓
Faire deux rotations vers l’arrière
↓
Attendre 500 ms
↓
```

```
Faire un son
```

Solution possible

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

ev3 = EV3Brick()

moteur = Motor(Port.A)

vitesse = 500

for i in range(3):

    moteur.run_angle(vitesse, 360)

    wait(500)

    moteur.run_angle(vitesse, -720)

    wait(500)

    ev3.speaker.beep()
```

41. Défi supplémentaire

Faire accélérer le moteur progressivement.

Le moteur doit utiliser les vitesses :

```
200
400
600
800
1000
```

Puis refaire le chemin inverse :

```
1000
800
600
400
200
```

Solution possible

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for vitesse in range(200, 1001, 200):

    moteur.run(vitesse)

    wait(500)
```

```
for vitesse in range(1000, 199, -200):
    moteur.run(vitesse)
    wait(500)
moteur.stop()
```

42. Questions de fin de cours

Question 1

À quoi sert une boucle `for` ?

Question 2

Combien de répétitions effectue :

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

Question 3

Quelles valeurs prend `i` dans :

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

Question 4

Que produit :

```
range(1, 6)
```

Question 5

Que signifie le troisième nombre dans :

```
range(0, 11, 2)
```

Question 6

Quelle boucle utiliser pour répéter exactement quatre fois une action ?

Question 7

Quelle boucle utiliser pour avancer jusqu'à ce qu'un capteur détecte quelque chose ?

Question 8

À quoi sert la variable `i` dans une boucle ?

Correction

Réponse 1

Une boucle `for` permet de répéter une série d'instructions, généralement un nombre déterminé de fois.

Réponse 2

```
5 répétitions
```

Réponse 3

```
0
1
2
3
4
```

Réponse 4

```
1
2
3
4
5
```

Réponse 5

Il indique le pas entre les différentes valeurs.

Dans :

```
range(0, 11, 2)
```

le programme avance de deux en deux.

Réponse 6

```
for
```

Réponse 7

```
while
```

Réponse 8

Elle permet notamment de connaître le numéro de la répétition et de faire varier des valeurs pendant la boucle.

Erreurs fréquentes

Erreur 1 : croire que `range(5)` va jusqu'à 5

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

affiche :

```
0
1
2
3
4
```

et non :

```
1
2
3
4
5
```

Erreur 2 : oublier les deux points

Incorrect :

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

Correct :

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

Erreur 3 : mauvaise indentation

Incorrect :

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

Correct :

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

Erreur 4 : mauvaise limite avec `range()`

Pour obtenir :

```
1
2
3
4
5
```

il faut écrire :

```
range(1, 6)
```

et non :

```
range(1, 5)
```

Cours 7 : Les moteurs EV3 avec MicroPython

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le rôle d'un moteur EV3
2. connecter et initialiser un moteur
3. faire tourner un moteur à une vitesse donnée
4. arrêter un moteur de différentes manières
5. faire tourner un moteur pendant une durée précise
6. faire tourner un moteur selon un angle précis
7. lire l'angle actuel d'un moteur
8. remettre son angle à zéro
9. comprendre la différence entre vitesse, durée et angle
10. utiliser des variables pour contrôler proprement un moteur

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 1 moteur moyen ou grand moteur EV3
3. 1 roue ou un élément permettant de visualiser la rotation
4. 1 ordinateur avec Visual Studio Code
5. MicroPython installé sur l'EV3
6. câble USB ou connexion permettant de transférer le programme

1. Introduction

Jusqu'à présent, nous avons surtout étudié les bases de Python :

```
variables

calculs

conditions

boucles while

boucles for
```

Nous avons déjà utilisé quelques commandes moteur.

Dans ce cours, nous allons comprendre précisément comment fonctionne un moteur EV3.

Tu peux utiliser le robot construit au **cours 3**.

Un moteur permet de transformer une commande informatique en mouvement.

On peut représenter cela ainsi :

```
PROGRAMME
  ↓
Commande moteur
  ↓
Rotation
  ↓
Mouvement mécanique
```

2. Les ports moteurs

La brique EV3 possède quatre ports destinés aux moteurs :

```
A

B

C

D
```

Ils se trouvent sur la partie supérieure de la brique EV3.

Pour utiliser un moteur dans un programme, nous devons indiquer sur quel port il est connecté.

Par exemple :

```
moteur = Motor(Port.A)
```

Cela signifie :

```
Le moteur utilisé par le programme est connecté au port A.
```

3. Importer les éléments nécessaires

Pour utiliser un moteur, nous avons besoin de :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
```

Puis nous créons notre moteur :

```
moteur = Motor(Port.A)
```

Programme minimal :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)
```

Le moteur est maintenant prêt à être utilisé.

4. Faire tourner un moteur

Pour faire tourner un moteur, nous pouvons utiliser :

```
moteur.run(500)
```

La valeur :

```
500
```

représente la vitesse du moteur en degrés par seconde.

Donc :

```
moteur.run(500)
```

signifie :

```
Faire tourner le moteur à 500 degrés par seconde.
```

5. Comprendre les degrés par seconde

Une rotation complète correspond à :

```
360 degrés
```

Si le moteur tourne à :

```
360 degrés par seconde
```

il effectue théoriquement :

```
1 rotation par seconde
```

À :

```
720 degrés par seconde
```

il effectue environ :

```
2 rotations par seconde
```

À :

```
180 degrés par seconde
```

il effectue environ :

```
une demi rotation par seconde
```

6. Premier programme

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

moteur.run(500)

wait(2000)

moteur.stop()
```

Le moteur tourne pendant :
`2 secondes`
puis s'arrête.

7. Exercice pratique 1

Modifier le programme pour faire tourner le moteur :

```
à 300 degrés par seconde

pendant 3 secondes
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

moteur.run(300)

wait(3000)

moteur.stop()
```

8. Utiliser des variables

Il est préférable d'éviter d'écrire directement les valeurs dans toutes les commandes.

Nous pouvons écrire :

```
vitesse = 500
duree = 2000
```

Puis :

```
moteur.run(vitesse)

wait(duree)

moteur.stop()
```

Programme complet :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 500
duree = 2000

moteur.run(vitesse)

wait(duree)

moteur.stop()
```

9. Changer le sens de rotation

Une vitesse positive fait tourner le moteur dans un sens.

```
moteur.run(500)
```

Une vitesse négative le fait tourner dans l'autre sens.

```
moteur.run(-500)
```

Exemple :

```
moteur.run(500)
wait(2000)

moteur.run(-500)
wait(2000)

moteur.stop()
```

Le moteur tourne :

```
2 secondes dans un sens

puis

2 secondes dans l'autre sens
```

10. Exercice pratique 2

Créer un programme qui réalise :

```
rotation avant à 400 degrés par seconde pendant 2 secondes

pause de 1 seconde

rotation arrière à 400 degrés par seconde pendant 2 secondes

arrêt
```

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 400

moteur.run(vitesse)
wait(2000)

moteur.stop()
wait(1000)

moteur.run(-vitesse)
wait(2000)

moteur.stop()
```

11. Plusieurs manières d'arrêter un moteur

Pybricks permet plusieurs comportements lors de l'arrêt d'un moteur.

Nous allons principalement utiliser :

```
moteur.stop()
moteur.brake()
```

et :

```
moteur.hold()
```

Ces commandes semblent similaires, mais leur comportement mécanique est différent.

12. `stop()`

```
moteur.stop()
```

coupe l'entraînement du moteur.

Le moteur peut encore continuer légèrement à tourner à cause de son inertie.

On peut imaginer cela comme :

```
Arrêter d'alimenter le moteur
```

13. `brake()`

```
moteur.brake()
```

utilise le moteur pour freiner.

Le moteur s'arrête généralement plus rapidement qu'avec `stop()`.

On peut imaginer cela comme :

```
Freiner le moteur
```

14. `hold()`

```
moteur.hold()
```

arrête le moteur et essaie ensuite de maintenir sa position.

Cela est utile lorsqu'un mécanisme ne doit plus bouger.

Par exemple :

```
bras robotique
```

```
pince
```

```
ascenseur
```

```
mécanisme devant rester dans une position précise
```

15. Expérience : comparer les arrêts

Faire tourner le moteur à une vitesse élevée.

Premier test :

```
moteur.run(800)
wait(2000)
moteur.stop()
```

Observer la manière dont le moteur s'arrête.

Deuxième test :

```
moteur.run(800)
wait(2000)
moteur.brake()
```

Observer la différence.

Troisième test :

```
moteur.run(800)
wait(2000)
moteur.hold()
```

Essayer ensuite de tourner l'axe à la main.

L'objectif est de faire ressentir physiquement les différences.

16. Exercice de réflexion

Quelle commande choisirais-tu pour :

Une roue d'un robot qui peut continuer légèrement sur son élan ?

```
stop()
```

Une roue qui doit s'arrêter plus rapidement ?

```
brake()
```

Un bras robotique qui doit rester en position ?

```
hold()
```

17. Faire tourner un moteur pendant une durée

Nous avons jusque-là utilisé :

```
moteur.run(vitesse)
wait(duree)
moteur.stop()
```

Il existe aussi une commande dédiée :

```
moteur.run_time(vitesse, duree)
```

Exemple :

```
moteur.run_time(500, 2000)
```

Cela signifie :

```
tourner à 500 degrés par seconde

pendant 2000 millisecondes
```

18. Comparaison

Première méthode :

```
moteur.run(500)
wait(2000)
moteur.stop()
```

Deuxième méthode :

```
moteur.run_time(500, 2000)
```

Les deux permettent d'obtenir un comportement proche.

La deuxième méthode est plus compacte lorsque nous voulons simplement faire tourner le moteur pendant une durée précise.

19. Exercice pratique 3

Faire tourner le moteur à :

```
600 degrés par seconde
```

pendant :

```
1,5 seconde
```

Solution :

```
moteur.run_time(600, 1500)
```

20. Faire tourner le moteur selon un angle

Nous pouvons demander au moteur de tourner d'un angle précis.

```
moteur.run_angle(500, 360)
```

Cela signifie :

```
tourner à 500 degrés par seconde

jusqu'à avoir effectué 360 degrés
```

Comme :

```
360 degrés = 1 rotation
```

le moteur effectue une rotation complète.

21. Quelques angles utiles

```
90 degrés = un quart de rotation

180 degrés = une demi rotation

360 degrés = une rotation

720 degrés = deux rotations

1080 degrés = trois rotations
```

22. Exercice pratique 4

Faire effectuer au moteur :

```
3 rotations complètes
```

à :

```
400 degrés par seconde
```

Calcul :

```
3 × 360 = 1080 degrés
```

Solution :

```
moteur.run_angle(400, 1080)
```

23. Utiliser une variable pour le nombre de rotations

Nous pouvons rendre le programme plus lisible :

```
rotations = 3

angle = rotations * 360

moteur.run_angle(400, angle)
```

Programme complet :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

rotations = 3
vitesse = 400

angle = rotations * 360

moteur.run_angle(vitesse, angle)
```

24. Faire tourner le moteur en sens inverse avec `run_angle()`

Nous pouvons utiliser un angle négatif :

```
moteur.run_angle(500, -360)
```

Le moteur effectue alors une rotation complète dans l'autre sens.

Exemple :

```
moteur.run_angle(500, 360)

moteur.run_angle(500, -360)
```

Le moteur revient approximativement à sa position de départ.

25. Lire l'angle du moteur

Le moteur EV3 possède un capteur de rotation intégré.

Nous pouvons lire sa position avec :

```
moteur.angle()
```

Exemple :

```
angle = moteur.angle()

print(angle)
```

Le résultat correspond à la position actuelle du moteur en degrés.

26. Expérience

Programme :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

while True:
    print(moteur.angle())
    wait(500)
```

Faire tourner doucement le moteur à la main.

Observer les valeurs affichées.

L'angle augmente dans un sens.

Il diminue dans l'autre.

27. Remettre l’angle à zéro

Nous pouvons remettre le compteur d’angle à zéro.

```
moteur.reset_angle(0)
```

Après cette instruction :

```
print(moteur.angle())
```

devrait afficher une valeur proche de :

```
0
```

28. Pourquoi remettre l’angle à zéro ?

Imaginons un bras robotique.

Au début du programme, nous voulons considérer sa position actuelle comme :

```
position 0
```

Nous pouvons écrire :

```
moteur.reset_angle(0)
```

Ensuite, une position de :

```
90 degrés
```

correspondra à un quart de rotation par rapport à cette position initiale.

29. Exercice pratique 5

Créer un programme qui :

```
remet l’angle du moteur à zéro

fait tourner le moteur de 720 degrés

affiche ensuite son angle
```

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)
moteur.reset_angle(0)
moteur.run_angle(500, 720)
print(moteur.angle())
```

Le résultat devrait être proche de :

```
720
```

30. Utiliser `while` avec l'angle du moteur

Nous pouvons maintenant combiner les notions précédentes.

Par exemple :

```
moteur.reset_angle(0)

moteur.run(500)

while moteur.angle() < 360:
    pass

moteur.stop()
```

Le moteur tourne jusqu'à atteindre environ :

```
360 degrés
```

Puis il s'arrête.

31. Que signifie `pass` ?

Dans :

```
while moteur.angle() < 360:
    pass
```

`pass` signifie simplement :

```
ne rien faire ici
```

La boucle continue donc à vérifier l'angle jusqu'à ce que la condition devienne fausse.

Cependant, pour cet objectif précis, il est plus simple d'utiliser :

```
moteur.run_angle(500, 360)
```

32. Pourquoi apprendre les deux méthodes ?

Parce que plus tard, nous voudrons parfois faire plusieurs choses pendant le mouvement.

Par exemple :

```
faire tourner le moteur

surveiller un capteur

afficher une valeur

vérifier une condition de sécurité

arrêter lorsqu'une certaine situation arrive
```

Dans ces cas, une boucle `while` devient très utile.

33. Utiliser une condition pendant le mouvement

Exemple :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

moteur.reset_angle(0)

moteur.run(500)

while moteur.angle() < 720:

    print(moteur.angle())

    wait(100)

moteur.stop()
```

Le programme affiche régulièrement la position du moteur pendant deux rotations.

34. Contrôler la vitesse avec une boucle

Nous pouvons faire accélérer progressivement le moteur.

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

for vitesse in range(100, 901, 100):

    moteur.run(vitesse)

    wait(500)

moteur.stop()
```

Le moteur passe par :

```
100
200
300
400
500
600
700
800
900
```

degrés par seconde.

35. Accélération puis décélération

```
for vitesse in range(100, 901, 100):
    moteur.run(vitesse)
    wait(300)
for vitesse in range(900, 99, -100):
    moteur.run(vitesse)
    wait(300)
moteur.stop()
```

Le moteur accélère puis ralentit progressivement.

36. Mini défi 1

Créer le comportement suivant :

```
2 rotations vers l'avant
↓
pause de 1 seconde
↓
2 rotations vers l'arrière
↓
arrêt
```

avec une vitesse de :

```
500 degrés par seconde
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 500
angle = 720

moteur.run_angle(vitesse, angle)

wait(1000)

moteur.run_angle(vitesse, -angle)

moteur.stop()
```

37. Mini défi 2

Créer un programme qui effectue :

```
5 fois
```

la séquence :

```
180 degrés vers l'avant

180 degrés vers l'arrière
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

for i in range(5):

    moteur.run_angle(400, 180)

    moteur.run_angle(400, -180)
```

38. Mini défi 3

Créer un programme qui affiche la position du moteur toutes les :

```
200 millisecondes
```

pendant que le moteur tourne.

Le programme doit s'arrêter lorsque l'angle dépasse :

```
1000 degrés
```

Une solution possible :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

moteur.reset_angle(0)

moteur.run(500)

while moteur.angle() < 1000:

    print(moteur.angle())

    wait(200)

moteur.stop()
```

39. Défi pratique : trouver une position

Place une barre LEGO sur l'axe du moteur.

Définisse sa position initiale comme :

```
0 degré
```

avec :

```
moteur.reset_angle(0)
```

Puis demande au moteur d'aller successivement vers :

```
90 degrés

180 degrés

270 degrés

360 degrés
```

Observe la position de la barre.

Une solution simple :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

moteur.reset_angle(0)

moteur.run_angle(300, 90)
wait(1000)

moteur.run_angle(300, 90)
wait(1000)

moteur.run_angle(300, 90)
wait(1000)

moteur.run_angle(300, 90)
```

40. Défi final : le moteur programmable

Créer un programme possédant les variables suivantes :

```
vitesse = 500
rotations = 3
pause = 1000
```

Le programme doit :

```
remettre l'angle à zéro
↓
effectuer 3 rotations vers l'avant
↓
afficher la position
↓
attendre 1 seconde
↓
effectuer 3 rotations vers l'arrière
↓
afficher la position finale
↓
maintenir le moteur en position
```

Solution possible

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur = Motor(Port.A)

vitesse = 500
rotations = 3
pause = 1000

angle = rotations * 360

moteur.reset_angle(0)

moteur.run_angle(vitesse, angle)

print(moteur.angle())

wait(pause)

moteur.run_angle(vitesse, -angle)

print(moteur.angle())

moteur.hold()
```

41. Tableau récapitulatif

Commande	Fonction
`Motor(Port.A)`	Créer un moteur connecté au port A
`run(vitesse)`	Faire tourner le moteur continuellement
`stop()`	Arrêter librement le moteur
`brake()`	Freiner le moteur
`hold()`	Arrêter et maintenir la position
`run_time(vitesse, temps)`	Tourner pendant une durée
`run_angle(vitesse, angle)`	Tourner d'un angle déterminé
`angle()`	Lire l'angle actuel
`reset_angle(0)`	Remettre l'angle à zéro

42. Questions de fin de cours

Question 1

À quoi correspondent les ports :

```
A
B
C
D
```

sur la brique EV3 ?

Question 2

Que signifie :

```
moteur.run(500)
```

Question 3

Combien de degrés correspondent à une rotation complète ?

Question 4

Quelle est la différence entre :

```
moteur.run(500)
```
et :
```
moteur.run_angle(500, 360)
```

Question 5

Comment faire tourner un moteur dans l'autre sens ?

Question 6

Quelle commande permet de lire la position actuelle du moteur ?

Question 7

À quoi sert :

```
moteur.reset_angle(0)
```

Question 8

Quelle différence principale existe entre :

```
stop()
brake()
```
et :
```
hold()
```

Correction

Réponse 1

Ils servent à connecter les moteurs à la brique EV3.

Réponse 2

Le moteur tourne à environ :

`500 degrés par seconde`

jusqu'à ce qu'une autre commande modifie son comportement.

Réponse 3

`360 degrés`

Réponse 4

`run()` fait tourner le moteur continuellement.

`run_angle()` lui demande de parcourir un angle déterminé.

Réponse 5

Utiliser une vitesse ou un angle négatif.

Par exemple :

```
moteur.run(-500)
```

Réponse 6

```
moteur.angle()
```

Réponse 7

Elle permet de considérer la position actuelle du moteur comme la nouvelle position zéro.

Réponse 8

`stop()` coupe l'entraînement.

`brake()` freine le moteur.

`hold()` arrête le moteur et maintient sa position.

Difficulté fréquente

Tu peux penser que :

```
moteur.run(500)
```

signifie :

```
faire 500 degrés
```

Ce n’est pas le cas.

Cela signifie :

```
500 degrés par seconde
```

La valeur correspond donc à une vitesse, pas à une distance de rotation.

Autre difficulté fréquente

Confondre :

```
run_time()
```

et :

```
run_angle()
```

Il faut revenir aux mots clés :

```
time = temps
```

```
angle = angle
```

Cours 8 : Contrôler deux moteurs et déplacer le robot EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. initialiser deux moteurs
2. identifier le moteur gauche et le moteur droit
3. faire avancer le robot
4. faire reculer le robot
5. arrêter les deux moteurs
6. tourner à gauche
7. tourner à droite
8. pivoter sur place
9. comprendre l'effet d'une différence de vitesse entre les deux roues
10. créer de petits parcours avec deux moteurs

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 2 roues identiques
4. 1 roulette ou roue libre
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. un espace libre au sol
8. du ruban adhésif pour matérialiser une ligne de départ

1. Introduction

Lors du cours précédent, nous avons appris à contrôler un moteur.

Par exemple :

```
moteur = Motor(Port.A)

moteur.run(500)
```

Avec un seul moteur, nous pouvons créer un mécanisme.

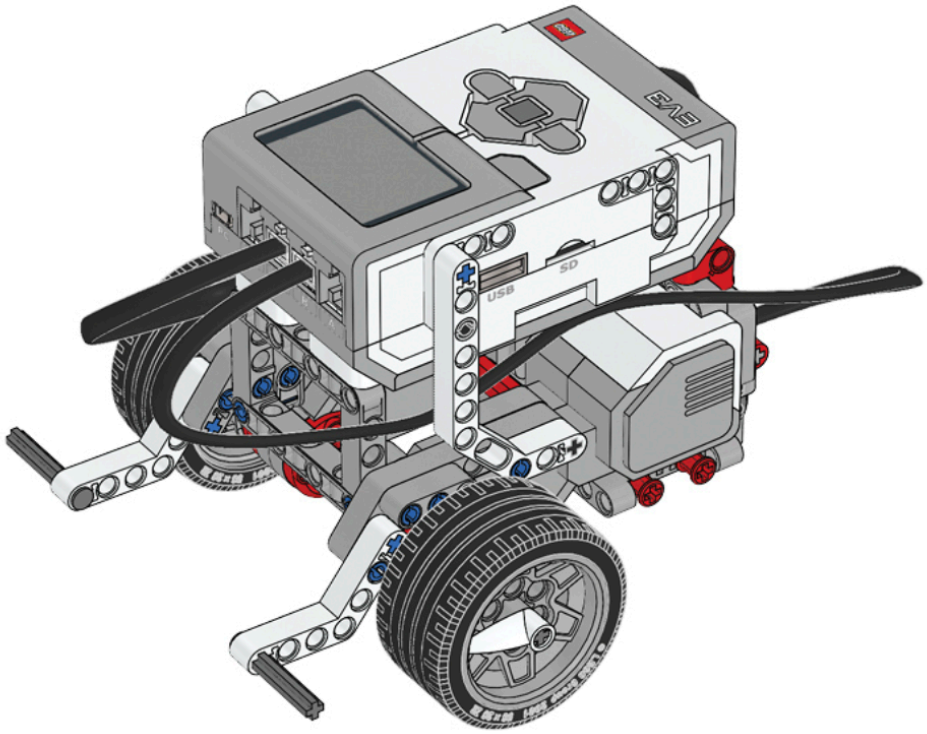
Mais pour créer un robot mobile capable de se déplacer, nous allons généralement utiliser deux moteurs.

Un pour la roue gauche.

Un pour la roue droite.

Construis le robot de base du document :

https://assets.education.lego.com/v3/assets/blt293eea581807678a/bltc8481dd2666822ff/5f8801e3f4f4cf0fa39d2fef/ev3-rem-driving-base.pdf?locale=en-us



On peut représenter le robot ainsi :

```
         AVANT

      [ ROBOT ]

   roue        roue
 gauche        droite

 moteur        moteur
 gauche        droit
```

2. Connexion des moteurs

Pour ce cours, nous utiliserons :

```
Port B = moteur gauche

Port C = moteur droit
```

Il est important que toute la classe utilise la même configuration.

Cela évite de devoir modifier les programmes d'un robot à l'autre.

3. Initialiser deux moteurs

Nous importons :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
```

Puis nous créons les deux moteurs :

```
moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)
```

Programme complet :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)
```

4. Faire avancer le robot

Pour avancer, les deux roues doivent tourner dans le sens qui pousse le robot vers l'avant.

Par exemple :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

Puis :

```
wait(2000)
```

Et enfin :

```
moteur_gauche.stop()
moteur_droit.stop()
```

Programme complet :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

moteur_gauche.run(500)
moteur_droit.run(500)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()
```

5. Attention au sens des moteurs

Selon la manière dont les moteurs sont montés, il est possible que le programme précédent fasse tourner une roue dans le mauvais sens.

Cela vient du fait que les deux moteurs sont souvent montés symétriquement.

Si le robot tourne sur lui même au lieu d'avancer, il faudra peut être utiliser :

```
moteur_gauche.run(500)
moteur_droit.run(-500)
```

Le sens exact dépend donc de la construction du robot.

Pour la suite du cours, nous supposerons que :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

fait avancer le robot.

Si ton montage nécessite un signe négatif sur un moteur, garde simplement cette adaptation dans tous les exercices.

6. Exercice pratique 1

Faire avancer le robot pendant :

```
3 secondes
```

à une vitesse de :

```
400 degrés par seconde
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

vitesse = 400
duree = 3000

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)

wait(duree)

moteur_gauche.stop()
moteur_droit.stop()
```

7. Faire reculer le robot

Pour reculer, nous pouvons utiliser une vitesse négative.

```
moteur_gauche.run(-500)
moteur_droit.run(-500)
```

Programme :

```
moteur_gauche.run(-500)
moteur_droit.run(-500)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()
```

8. Utiliser une variable

Nous pouvons écrire :

```
vitesse = 500
```

Puis :

```
moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)
```

Pour reculer :

```
moteur_gauche.run(-vitesse)
moteur_droit.run(-vitesse)
```

Cela évite de créer une deuxième variable inutile.

9. Exercice pratique 2

Faire effectuer au robot :

```
Avancer pendant 2 secondes

Pause pendant 1 seconde

Reculer pendant 2 secondes

Arrêt
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

vitesse = 400
```

```
moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()

wait(1000)

moteur_gauche.run(-vitesse)
moteur_droit.run(-vitesse)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()
```

10. Comment tourner ?

Pour tourner, les deux roues ne doivent plus forcément tourner à la même vitesse.

C'est une notion extrêmement importante.

Lorsque les deux roues tournent à la même vitesse :

```
gauche = 500

droite = 500
```

le robot avance tout droit.

Lorsque les vitesses sont différentes :

```
gauche = 200

droite = 500
```

le robot tourne.

Pourquoi ?

La roue droite parcourt davantage de distance que la roue gauche.

Le robot suit donc une courbe.

11. Tourner vers la gauche

Pour tourner progressivement vers la gauche, la roue gauche doit tourner plus lentement.

Par exemple :

```
moteur_gauche.run(200)
moteur_droit.run(500)
```

Le côté droit avance plus vite.

Le robot tourne donc vers la gauche.

On peut représenter cela ainsi :

```
GAUCHE              DROITE

200                  500

lent                 rapide
```

12. Tourner vers la droite

On fait l'inverse :

```
moteur_gauche.run(500)
moteur_droit.run(200)
```

La roue gauche avance plus rapidement.

Le robot tourne donc vers la droite.

13. Exercice pratique 3

Tester successivement :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

puis :

```
moteur_gauche.run(200)
moteur_droit.run(500)
```

puis :

```
moteur_gauche.run(500)
moteur_droit.run(200)
```

Observer les trajectoires.

Compléter le tableau.

Vitesse gauche	Vitesse droite	Trajectoire
500	500	
200	500	
500	200	
0	500	
500	0	

14. Tourner autour d’une roue

Nous pouvons complètement arrêter une roue.

Par exemple :

```
moteur_gauche.run(0)
moteur_droit.run(500)
```

Le robot tourne autour de sa roue gauche.

Nous pourrions aussi simplement arrêter le moteur gauche :

```
moteur_gauche.stop()
moteur_droit.run(500)
```

Le rayon du virage devient beaucoup plus petit.

15. Tourner sur place

Nous pouvons aller encore plus loin.

Une roue tourne vers l’avant.

L’autre tourne vers l’arrière.

```
moteur_gauche.run(-500)
moteur_droit.run(500)
```

Le robot pivote alors presque autour de son centre.

On appelle souvent cela :

```
rotation sur place
```

ou :

```
pivot
```

16. Comparaison des trois types de virages

Virage large

```
moteur_gauche.run(300)
moteur_droit.run(500)
```

Les deux roues avancent.

Le robot décrit une courbe.

Virage serré

```
moteur_gauche.stop()
moteur_droit.run(500)
```

Une roue reste immobile.

Pivot sur place

```
moteur_gauche.run(-500)
moteur_droit.run(500)
```

Les roues tournent dans des directions opposées.

17. Expérience

Créer trois zones au sol.

Tester :

```
Virage large

Virage serré

Pivot
```

Observer :

```
le rayon du virage

la vitesse du robot

la précision de la rotation
```

D'après toi :

Quel type de rotation serait le plus adapté dans un espace très étroit ?

Réponse :

```
Le pivot sur place.
```

18. Utiliser `run_angle()` avec deux moteurs

Nous pouvons également faire tourner les deux moteurs d'un angle précis.

```
moteur_gauche.run_angle(500, 720)
moteur_droit.run_angle(500, 720)
```

Attention toutefois, ces instructions sont exécutées l'une après l'autre.

Le premier moteur termine donc normalement son mouvement avant que le second commence.

Ce n'est pas ce que nous voulons pour faire avancer correctement un robot.

19. Pourquoi ce programme pose problème ?

```
moteur_gauche.run_angle(500, 720)
moteur_droit.run_angle(500, 720)
```

Le programme fait :

```
Moteur gauche

↓

720 degrés

↓

fin

↓

Moteur droit

↓

720 degrés
```

Le robot ne se déplace donc pas correctement.

Nous devons lancer les mouvements de manière adaptée.

Dans les prochaines leçons, nous utiliserons notamment `DriveBase`, qui simplifiera énormément le contrôle coordonné des deux moteurs.

20. Contrôle avec `run()`

Pour l'instant, l'utilisation de :

```
moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)
```

est très utile car les deux moteurs continuent à tourner pendant que le programme poursuit son exécution.

Puis nous pouvons attendre :

```
wait(2000)
```

avant de les arrêter.

21. Le robot ne va pas parfaitement droit

Même avec :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

le robot peut légèrement dévier.

Pourquoi ?

Parce qu'un robot réel n'est jamais parfaitement symétrique.

Quelques causes possibles :

```
différence entre les moteurs

frottements différents

roues légèrement différentes

poids mal réparti

sol irrégulier

jeu mécanique

niveau de batterie
```

22. Expérience de précision

Tracer une ligne droite de :

```
2 mètres
```

Faire avancer le robot avec :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

Observer où il termine.

Faire trois essais.

Essai	Distance	Déviation à gauche ou droite
1	2 m	
2	2 m	
3	2 m	

23. Corriger légèrement une trajectoire

Supposons que le robot tourne constamment légèrement vers la droite.

Nous pouvons essayer :

```
moteur_gauche.run(480)
moteur_droit.run(500)
```

ou l'inverse selon le comportement observé.

Nous effectuons alors une calibration.

C’est un principe important en robotique :

```
Programmer

Tester

Mesurer

Corriger

Tester à nouveau
```

24. Utiliser deux variables de vitesse

Nous pouvons écrire :

```
vitesse_gauche = 500
vitesse_droite = 500
```

Puis :

```
moteur_gauche.run(vitesse_gauche)
moteur_droit.run(vitesse_droite)
```

Cela permettra plus tard de modifier facilement la direction du robot.

25. Exemple de virage

```
vitesse_gauche = 200
vitesse_droite = 500

moteur_gauche.run(vitesse_gauche)
moteur_droit.run(vitesse_droite)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()
```

26. Exercice pratique 4

Créer trois programmes.

Programme A

Faire avancer le robot tout droit.

```
gauche = 400
droite = 400
```

Programme B

Faire tourner le robot vers la gauche.

```
gauche = 200
droite = 500
```

Programme C

Faire tourner le robot vers la droite.

```
gauche = 500
droite = 200
```

Comparer les trajectoires.

27. Créer une fonction d'arrêt

Nous n'avons pas encore étudié les fonctions en détail, mais voici une idée que nous utiliserons plus tard.

Au lieu d'écrire continuellement :

```
moteur_gauche.stop()
moteur_droit.stop()
```

nous pourrons créer :

```
def arreter():
    moteur_gauche.stop()
    moteur_droit.stop()
```

Puis écrire simplement :

```
arreter()
```

Nous étudierons ce principe en détail prochainement.

28. Faire une séquence de déplacement

Créons maintenant un petit parcours.

```
Avancer 2 secondes

Tourner à droite 1 seconde

Avancer 2 secondes

Arrêt
```

Programme :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

vitesse = 400

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)

wait(2000)
```

```
moteur_gauche.run(vitesse)
moteur_droit.run(-vitesse)

wait(1000)

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()
```

29. Comprendre la rotation

Lors de :

```
moteur_gauche.run(vitesse)
moteur_droit.run(-vitesse)
```

les roues tournent dans des sens opposés.

Le robot pivote donc sur place.

La durée :

```
wait(1000)
```

détermine ici approximativement l'angle de rotation.

Mais cette méthode n'est pas très précise.

Si nous changeons :

```
la batterie

le sol

le poids

la vitesse
```

l'angle obtenu peut changer.

Nous verrons prochainement de meilleures méthodes.

30. Exercice : trouver 90 degrés

Faire pivoter le robot avec :

```
moteur_gauche.run(300)
moteur_droit.run(-300)
```

Tester différentes durées.

Par exemple :

```
300 ms

500 ms

700 ms

900 ms

1100 ms
```

Chercher la durée permettant d'obtenir approximativement :

```
90 degrés
```

Noter le résultat.

```
Durée obtenue pour environ 90° :

_________________ ms
```

31. Pourquoi cette méthode n'est-elle pas idéale ?

Parce que nous contrôlons la rotation avec le temps.

Le robot tourne pendant une certaine durée mais nous ne mesurons pas directement son angle.

Un changement mécanique peut donc modifier le résultat.

Plus tard, nous utiliserons :

```
les encodeurs moteurs

DriveBase

le gyroscope
```

pour obtenir une meilleure précision.

32. Utiliser une boucle `for`

Nous pouvons maintenant utiliser les notions précédentes pour créer des parcours répétitifs.

Imaginons que nous voulions effectuer quatre fois :

```
Avancer

Puis tourner
```

Nous pouvons écrire :

```
for i in range(4):

    moteur_gauche.run(400)
    moteur_droit.run(400)

    wait(1000)

    moteur_gauche.run(400)
    moteur_droit.run(-400)

    wait(500)
```

Cette structure peut servir à créer un carré approximatif.

33. Premier carré approximatif

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

vitesse = 400
temps_avance = 1500
temps_rotation = 600

for i in range(4):

    moteur_gauche.run(vitesse)
    moteur_droit.run(vitesse)

    wait(temps_avance)

    moteur_gauche.run(vitesse)
    moteur_droit.run(-vitesse)

    wait(temps_rotation)

moteur_gauche.stop()
moteur_droit.stop()
```

Il faudra ajuster :

```
temps_rotation
```

pour obtenir environ 90 degrés.

34. Défi pratique 1 : le carré

Faire suivre au robot une trajectoire aussi proche que possible d'un carré.

Chaque côté doit mesurer environ :

```
50 cm
```

Tu peux modifier :

```
vitesse = 400
temps_avance = 1500
temps_rotation = 600
```

mais conserver :

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

35. Défi pratique 2 : le triangle

Créer une trajectoire approximativement triangulaire.

Il faudra répéter :

```
Avancer

Tourner
```

trois fois.

Structure :

```
for i in range(3):
```

Question :

Quel angle extérieur faut il approximativement utiliser pour un triangle équilatéral ?

Réponse :

```
120 degrés
```

36. Défi pratique 3 : marche avant et arrière

Créer le parcours suivant :

```
Avancer 2 secondes
↓
Reculer 1 seconde
↓
Tourner à droite
↓
Avancer 2 secondes
↓
Arrêt
```

Tu dois réutiliser :

```
variables

vitesses positives

vitesses négatives

wait()
```

37. Défi pratique 4 : slalom

Placer trois objets sur le sol.

Le robot doit approximativement effectuer :

```
courbe gauche

courbe droite

courbe gauche

courbe droite
```

Pour une courbe gauche :

```
moteur_gauche.run(200)
moteur_droit.run(500)
```

Pour une courbe droite :

```
moteur_gauche.run(500)
moteur_droit.run(200)
```

Adapter les durées pour contourner les obstacles.

38. Défi final : parcours complet

Créer le parcours suivant :

```
DÉPART
↓
Avancer
↓
Virage à droite
↓
Avancer
↓
Virage à gauche
↓
Avancer
↓
Pivot de 180 degrés
↓
Retour en arrière
↓
STOP
```

Tu dois utiliser au minimum :

```
2 variables de vitesse

1 boucle

1 vitesse négative

1 pivot

1 virage courbe
```

39. Exemple de solution

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

vitesse = 400
vitesse_lente = 150

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)
wait(1500)

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse_lente)
wait(1000)

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)
wait(1500)

moteur_gauche.run(vitesse_lente)
moteur_droit.run(vitesse)
wait(1000)

moteur_gauche.run(vitesse)
moteur_droit.run(vitesse)
wait(1500)

moteur_gauche.run(vitesse)
moteur_droit.run(-vitesse)
wait(1200)

for i in range(2):

    moteur_gauche.run(vitesse)
    moteur_droit.run(vitesse)
    wait(1000)

moteur_gauche.stop()
moteur_droit.stop()
```

Les durées devront être adaptées au robot utilisé.

40. Tableau récapitulatif

Moteur gauche	Moteur droit	Mouvement approximatif
500	500	Avancer
0	0	Arrêt
−500	−500	Reculer
200	500	Courbe vers la gauche
500	200	Courbe vers la droite
0	500	Rotation autour de la roue gauche
500	0	Rotation autour de la roue droite
−500	500	Pivot dans un sens
500	−500	Pivot dans l'autre sens

41. Questions de fin de cours

Question 1

Quels ports utilisons nous pour les moteurs gauche et droit ?

Question 2

Comment faire avancer un robot à deux roues ?

Question 3

Comment faire reculer le robot ?

Question 4

Que se passe t il si la roue gauche tourne moins vite que la roue droite ?

Question 5

Comment faire pivoter le robot sur place ?

Question 6

Pourquoi le robot ne va t il pas forcément parfaitement droit lorsque les deux moteurs reçoivent la même vitesse ?

Question 7

Quelle différence existe entre un virage large et un pivot ?

Question 8

Pourquoi une rotation basée uniquement sur `wait()` manque t elle de précision ?

Correction

Réponse 1

Dans notre configuration :

```
B = moteur gauche
```

```
C = moteur droit
```

Réponse 2

Faire tourner les deux moteurs à la même vitesse dans le sens correspondant à l'avant.

Réponse 3

Utiliser une vitesse négative sur les deux moteurs.

Réponse 4

Le robot tourne vers la gauche.

Réponse 5

Faire tourner les deux roues dans des directions opposées.

Par exemple :

```
moteur_gauche.run(-500)
moteur_droit.run(500)
```

Réponse 6

À cause des différences mécaniques, des frottements, des moteurs, du sol et d'autres imperfections physiques.

Réponse 7

Pendant un virage large, les deux roues avancent à des vitesses différentes.

Pendant un pivot, elles tournent dans des directions opposées.

Réponse 8

Parce que la durée ne mesure pas directement l'angle réellement effectué par le robot.

Difficulté fréquente

Tu pourrais penser que :

```
moteur gauche plus rapide
=
tourner à gauche
```

Alors que c'est généralement l'inverse.

Si la roue gauche avance davantage, le robot tourne vers la droite.

Une démonstration physique est beaucoup plus efficace qu'une simple explication.

Autre difficulté fréquente

Les deux moteurs peuvent être montés dans des orientations différentes.

Si un robot tourne sur place avec :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

cela signifie qu'il faut inverser le sens logique de l'un des moteurs selon le montage.

Il vaut mieux corriger cela dès le début de la leçon et garder une configuration identique pour tous les exercices.

Cours 9 : Utiliser `DriveBase` pour déplacer précisément le robot EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le rôle de `DriveBase`
2. créer un robot à partir de deux moteurs
3. configurer le diamètre des roues
4. configurer l'écartement des roues
5. avancer d'une distance précise
6. reculer d'une distance précise
7. tourner d'un angle précis
8. utiliser `drive()` pour déplacer le robot continuellement
9. arrêter correctement le robot
10. lire la distance et l'angle parcourus
11. calibrer le robot
12. programmer des formes géométriques simples

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 2 roues identiques
4. 1 roulette ou point d'appui
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. une règle ou un mètre
8. du ruban adhésif pour tracer des repères au sol

1. Rappel du cours précédent

Lors du cours précédent, nous contrôlions directement les deux moteurs.

Par exemple :

```
moteur_gauche.run(500)
moteur_droit.run(500)
```

pour avancer.

Ou :

```
moteur_gauche.run(500)
moteur_droit.run(-500)
```

pour pivoter.

Cette méthode fonctionne, mais elle nous oblige à réfléchir constamment à la vitesse de chaque roue.

Pour avancer d'une distance précise, nous devions également utiliser des calculs ou une durée approximative.

Nous allons maintenant utiliser une solution beaucoup plus pratique :

`DriveBase`

2. Qu'est-ce que `DriveBase` ?

`DriveBase` permet de représenter un robot possédant deux roues motrices.

Au lieu de contrôler séparément :

```
moteur gauche
```

```
et
```

```
moteur droit
```

nous pouvons contrôler directement :

```
le robot
```

Par exemple :

```
robot.straight(500)
```

signifie :

```
Avancer de 500 mm
```

Et :

```
robot.turn(90)
```

signifie :

```
Tourner de 90 degrés
```

Le programme devient donc beaucoup plus simple à lire.

3. Importer `DriveBase`

Nous devons importer :

```
from pybricks.robotics import DriveBase
```

Nous avons également besoin des moteurs :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
```

4. Créer les moteurs

Nous utilisons toujours :

```
moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)
```

Puis nous créons le robot.

```
robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

5. Comprendre les paramètres

Deux mesures sont très importantes :

```
wheel_diameter
```

et :

```
axle_track
```

wheel_diameter

Cela correspond au diamètre des roues.

Dans notre exemple :

```
wheel_diameter=56
```

signifie :

```
diamètre de roue = 56 mm
```

axle_track

Cela représente approximativement la distance entre les points de contact des roues gauche et droite.

Exemple :

```
axle_track=114
```

signifie :

```
écartement des roues = 114 mm
```

Cette valeur est particulièrement importante pour calculer les rotations du robot.

6. Pourquoi ces mesures sont-elles nécessaires ?

Pour avancer de :

```
500 mm
```

le robot doit calculer combien de rotations les roues doivent effectuer.

Pour tourner de :

```
90 degrés
```

le robot doit calculer la distance que chaque roue doit parcourir.

Il a donc besoin de connaître :

```
le diamètre des roues

et

l'écartement des roues
```

7. Programme de base

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

robot.straight(500)
```

Le robot doit avancer d'environ :

```
500 mm
```

soit :

```
50 cm
```

8. Première expérience

Tracer deux lignes espacées de :

```
500 mm
```

Placer le robot sur la première ligne.

Lancer :

```
robot.straight(500)
```

Mesurer la position finale.

Compléter :

Essai	Distance demandée	Distance réelle
1	500 mm	
2	500 mm	
3	500 mm	

9. Avancer

Pour avancer :

```
robot.straight(500)
```

La distance est exprimée en :

```
millimètres
```

Quelques exemples :

```
robot.straight(100)
```

correspond à :

```
10 cm
robot.straight(500)
```

correspond à :

```
50 cm
robot.straight(1000)
```

correspond à :

```
1 mètre
```

10. Reculer

Pour reculer, il suffit d'utiliser une distance négative.

```
robot.straight(-500)
```

Le robot recule de :

```
500 mm
```

11. Exercice pratique 1

Créer un programme qui réalise :

```
Avancer de 300 mm

Puis

Reculer de 300 mm
```

Solution :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

robot.straight(300)

robot.straight(-300)
```

Le robot devrait revenir approximativement à son point de départ.

12. Tourner avec `turn()`

Pour tourner, nous pouvons utiliser :

```
robot.turn(90)
```

Le robot effectue une rotation d'environ :

```
90 degrés
```

Pour tourner dans l'autre sens :

```
robot.turn(-90)
```

13. Sens de rotation

Dans la configuration habituelle :

```
robot.turn(90)
```

fait tourner le robot dans un sens.

Et :

```
robot.turn(-90)
```

dans l'autre sens.

Il est utile de tester le robot réel et de noter :

```
angle positif = _____
angle négatif = _____
```

Selon la configuration, cela te permet de savoir clairement quel signe utiliser.

14. Expérience sur les angles

Faire tester :

```
robot.turn(45)
```

Puis :

```
robot.turn(90)
```

Puis :

```
robot.turn(180)
```

Puis :

```
robot.turn(360)
```

Observer les rotations.

15. Repères utiles

```
45° = huitième de tour
90° = quart de tour
180° = demi-tour
360° = tour complet
```

16. Exercice pratique 2

Programmer :

```
Avancer de 500 mm

Tourner de 90°

Avancer de 500 mm
```

Solution :

```
robot.straight(500)

robot.turn(90)

robot.straight(500)
```

17. Le carré

Nous pouvons maintenant réaliser un carré beaucoup plus simplement.

Un carré possède :

```
4 côtés
```

et chaque changement de direction correspond à :

```
90 degrés
```

Programme :

```
for i in range(4):

    robot.straight(500)

    robot.turn(90)
```

Le robot doit parcourir un carré de :

```
50 cm de côté
```

18. Programme complet du carré

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

```
for i in range(4):

    robot.straight(500)

    robot.turn(90)
```

19. Exercice pratique 3

Créer un rectangle de :

```
600 mm de longueur

300 mm de largeur
```

Une solution possible :

```
for i in range(2):

    robot.straight(600)
    robot.turn(90)

    robot.straight(300)
    robot.turn(90)
```

20. Le triangle équilatéral

Pour un triangle équilatéral, le robot doit effectuer trois côtés.

```
for i in range(3):
```

L'angle de rotation extérieur est :

```
120 degrés
```

Programme :

```
for i in range(3):

    robot.straight(500)

    robot.turn(120)
```

21. Exercice de réflexion

Pourquoi utilisons nous :
```
120 degrés
```

et non :
```
60 degrés
```

alors qu'un triangle équilatéral possède des angles internes de 60 degrés ?

Parce que le robot doit effectuer l'angle extérieur nécessaire pour changer sa direction.

```
180° moins 60° = 120°
```

22. Créer d’autres polygones

Pour un polygone régulier, l’angle de rotation extérieur peut être calculé par :

```
360 / nombre de côtés
```

Par exemple, pour un carré :

```
360 / 4 = 90°
```

Pour un triangle :

```
360 / 3 = 120°
```

Pour un hexagone :

```
360 / 6 = 60°
```

23. Programme automatique pour un polygone

```
cotes = 6

angle = 360 / cotes

for i in range(cotes):

    robot.straight(300)

    robot.turn(angle)
```

Le robot réalise ici un hexagone.

24. Défi pratique 1

Créer une variable :

```
cotes = 5
```

Puis calculer automatiquement l’angle.

Le robot doit réaliser un pentagone.

Solution :

```
cotes = 5
distance = 300

angle = 360 / cotes

for i in range(cotes):

    robot.straight(distance)

    robot.turn(angle)
```

25. Pourquoi le robot ne revient-il pas parfaitement au point de départ ?

Même si les calculs sont corrects, le robot réel peut accumuler de petites erreurs.

Par exemple :

```
premier virage = 89°
deuxième = 91°
troisième = 90°
quatrième = 89°
```

Les petites erreurs peuvent s'additionner.

C'est ce que l'on appelle souvent :

```
une erreur cumulative
```

26. Calibration de `wheel_diameter`

Supposons que nous demandions :

```
robot.straight(1000)
```

mais que le robot parcoure seulement :

```
950 mm
```

Cela peut signifier que la valeur utilisée pour le diamètre des roues n'est pas parfaitement adaptée au robot réel.

On peut alors ajuster :

```
wheel_diameter=56
```

en fonction des mesures obtenues.

27. Calibration de `axle_track`

Supposons que nous demandions :
`robot.turn(360)`

mais que le robot ne réalise pas exactement un tour complet.

Nous pouvons modifier :
`axle_track=114`
jusqu'à obtenir une rotation plus précise.

`axle_track` influence donc surtout la précision des rotations.

28. Expérience de calibration de distance

Faire :

```
robot.straight(1000)
```

Mesurer la distance réelle.

Effectuer trois essais.

Essai	Demandé	Réel
1	1000 mm	
2	1000 mm	
3	1000 mm	

Calculer ensuite une moyenne.

29. Expérience de calibration de rotation

Tracer une ligne indiquant l'orientation initiale du robot.

Exécuter :

```
robot.turn(360)
```

Le robot doit revenir dans la même orientation.

Faire trois essais.

Essai	Angle demandé	Observation
1	360°	
2	360°	
3	360°	

Ajuster `axle_track` si nécessaire.

30. Déplacement continu avec `drive()`

`straight()` effectue une distance puis attend que le déplacement soit terminé.

Mais nous pouvons aussi demander au robot de rouler continuellement.

```
robot.drive(200, 0)
```

Les deux paramètres représentent :

```
vitesse de déplacement
```

```
vitesse de rotation
```

31. Premier paramètre de `drive()`

Dans :

```
robot.drive(200, 0)
```

`200` correspond à la vitesse linéaire du robot.

Elle est exprimée en :

```
mm/s
```

Donc le robot essaie d'avancer à :

```
200 mm/s
```

soit environ :

```
20 cm/s
```

32. Deuxième paramètre de `drive()`

Le deuxième paramètre représente la vitesse de rotation.

```
robot.drive(200, 0)
```

signifie :

```
avancer tout droit
```

car :

```
rotation = 0
```

Si nous écrivons :

```
robot.drive(200, 45)
```

le robot avance tout en tournant.

Il suit donc une courbe.

33. Exemple avec `drive()`

```
from pybricks.tools import wait

robot.drive(200, 0)

wait(2000)

robot.stop()
```

Le robot avance pendant deux secondes puis s'arrête.

34. Comparer `straight()` et `drive()`

Avec :

```
robot.straight(500)
```

nous demandons :

```
une distance précise
```

Avec :

```
robot.drive(200, 0)
```

nous demandons :

```
une vitesse
```

Le robot continue jusqu'à ce qu'une autre instruction modifie son comportement.

35. Exemple de courbe

```
robot.drive(200, 50)

wait(3000)

robot.stop()
```

Le robot se déplace en suivant une trajectoire courbe.

Tester également :

```
robot.drive(200, 100)
```

Puis :

```
robot.drive(200, 200)
```

Observer la différence.

36. Exercice pratique 4

Tester :

```
robot.drive(200, 0)
robot.drive(200, 50)
robot.drive(200, 100)
robot.drive(200, -100)
```

Compléter :

Vitesse	Rotation	Mouvement observé
200	0	
200	50	
200	100	
200	−100	

37. Faire pivoter avec `drive()`

Si la vitesse d'avancement vaut :

```
0
```

mais que la vitesse de rotation n'est pas nulle :

```
robot.drive(0, 90)
```

le robot tourne sur place.

Par exemple :

```
robot.drive(0, 90)
wait(2000)
robot.stop()
```

38. Arrêter le robot

Pour arrêter `DriveBase`, nous pouvons utiliser :

```
robot.stop()
```

Il est important de ne pas oublier cette instruction lorsqu'on utilise :

```
robot.drive()
```

car `drive()` lance un mouvement continu.

39. Lire la distance parcourue

`DriveBase` peut également suivre la distance parcourue.

Nous pouvons utiliser :

```
robot.distance()
```

Exemple :

```
print(robot.distance())
```

40. Lire l’angle du robot

Nous pouvons également utiliser :

```
robot.angle()
```

Cela donne une estimation de l’angle de rotation accumulé par le robot.

Exemple :

```
robot.turn(90)

print(robot.angle())
```

Le résultat doit être proche de :

```
90
```

41. Remettre les mesures à zéro

Nous pouvons remettre les mesures de `DriveBase` à zéro avec :

```
robot.reset()
```

Après :

```
robot.reset()
```

les valeurs de :

```
robot.distance()
```

et :

```
robot.angle()
```

repartent de zéro.

42. Expérience

Créer :

```
robot.reset()

robot.straight(500)

robot.turn(90)

print(robot.distance())
print(robot.angle())
```

Observer les valeurs.

Puis ajouter :

```
robot.reset()
```

et afficher de nouveau les mesures.

43. Utiliser `while` avec `DriveBase`

Nous pouvons maintenant combiner `drive()` et une boucle.

Exemple :

```
robot.reset()

robot.drive(200, 0)

while robot.distance() < 500:
    pass

robot.stop()
```

Le robot avance jusqu'à ce que la distance mesurée atteigne environ :

```
500 mm
```

44. Pourquoi utiliser cette méthode ?

Pour simplement avancer de 500 mm, nous préférerons :

```
robot.straight(500)
```

Mais la boucle devient intéressante si nous devons effectuer d'autres actions pendant le déplacement.

Par exemple :

```
observer un capteur

afficher la distance

modifier la vitesse

vérifier une condition
```

45. Exemple avec affichage de la distance

```
from pybricks.tools import wait

robot.reset()

robot.drive(200, 0)

while robot.distance() < 500:

    print(robot.distance())

    wait(100)

robot.stop()
```

Le programme affiche régulièrement la distance parcourue.

46. Défi pratique 2 : aller-retour

Créer le parcours suivant :

```
Avancer de 1 mètre

↓

Tourner de 180 degrés

↓

Avancer de 1 mètre

↓

Arrêt
```

Solution :

```
robot.straight(1000)

robot.turn(180)

robot.straight(1000)
```

47. Défi pratique 3 : parcours en U

Créer :

```
Avancer 500 mm

Tourner 90°

Avancer 300 mm

Tourner 90°

Avancer 500 mm
```

Le robot doit dessiner une forme proche d'un U.

48. Défi pratique 4 : escalier

Créer une trajectoire :

```
300 mm
90°
300 mm
moins 90°
300 mm
90°
300 mm
moins 90°
300 mm
```

Le robot doit créer une trajectoire en escalier.

Il est intéressant d'utiliser une boucle.

49. Exemple d'escalier

```
for i in range(2):
    robot.straight(300)
    robot.turn(90)
    robot.straight(300)
    robot.turn(-90)
robot.straight(300)
```

50. Défi final : parcours géométrique

Créer un programme capable de réaliser successivement :

```
un carré
puis
un triangle
puis
un demi tour
puis
retour au point de départ
```

Tu dois utiliser :

```
DriveBase

straight()

turn()

for

variables
```

51. Exemple de solution

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

distance_carre = 300

for i in range(4):

    robot.straight(distance_carre)

    robot.turn(90)

robot.straight(300)

for i in range(3):

    robot.straight(300)

    robot.turn(120)

robot.turn(180)

robot.straight(300)
```

Les dimensions peuvent être adaptées au terrain disponible.

52. Tableau récapitulatif

Commande	Fonction
`DriveBase(...)`	Créer le robot roulant
`straight(500)`	Avancer de 500 mm
`straight(-500)`	Reculer de 500 mm
`turn(90)`	Tourner d'environ 90°
`turn(-90)`	Tourner dans l'autre sens
`drive(200, 0)`	Rouler continuellement à 200 mm/s
`drive(200, 50)`	Avancer en décrivant une courbe
`drive(0, 90)`	Pivoter sur place
`stop()`	Arrêter le robot
`distance()`	Lire la distance parcourue
`angle()`	Lire l'angle accumulé
`reset()`	Remettre les mesures à zéro

53. Questions de fin de cours

Question 1

À quoi sert `DriveBase` ?

Question 2

Que représente :

```
wheel_diameter
```

Question 3

Que représente :

```
axle_track
```

Question 4

Que fait :

```
robot.straight(500)
```

Question 5

Comment reculer de 500 mm ?

Question 6

Que fait :

```
robot.turn(90)
```

Question 7

Quelle différence existe entre :

```
robot.straight(500)
```

et :

```
robot.drive(500, 0)
```

Question 8

À quoi sert :

```
robot.stop()
```

Question 9

Comment lire la distance parcourue ?

Question 10

Pourquoi faut-il calibrer `wheel_diameter` et `axle_track` ?

Correction

Réponse 1

`DriveBase` permet de contrôler les deux moteurs comme un seul robot roulant.

Réponse 2

Le diamètre des roues en millimètres.

Réponse 3

L'écartement utilisé pour calculer les rotations du robot.

Réponse 4

Le robot avance d'environ 500 mm.

Réponse 5

```
robot.straight(-500)
```

Réponse 6

Le robot effectue une rotation d'environ 90 degrés.

Réponse 7

`straight()` demande une distance.

`drive()` demande une vitesse de déplacement et une vitesse de rotation.

Réponse 8

Elle arrête le déplacement continu du robot.

Réponse 9

```
robot.distance()
```

Réponse 10

Parce que les dimensions et le comportement du robot réel influencent la précision des déplacements et des rotations.

Erreur fréquente

Confondre :

```
robot.straight(500)
```

avec :

```
robot.drive(500, 0)
```

Le premier utilise :

```
une distance en mm
```

Le deuxième utilise :

```
une vitesse en mm/s
def avancer():
    robot.straight(500)
```

Puis utiliser simplement :

```
avancer()
```

Nous irons ensuite plus loin avec des paramètres :

```
def avancer(distance):
    robot.straight(distance)
```

pour pouvoir écrire :

```
avancer(500)
avancer(1000)
avancer(250)
```

Ce sera une étape essentielle pour créer des programmes robotiques plus propres, plus lisibles et beaucoup plus faciles à réutiliser.

Cours 10 : Les fonctions avec MicroPython et LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre ce qu'est une fonction
2. créer une fonction avec `def`
3. appeler une fonction
4. comprendre l'intérêt de réutiliser du code
5. utiliser des paramètres
6. créer plusieurs fonctions pour contrôler le robot
7. utiliser une fonction dans une boucle
8. comprendre la notion de valeur de retour
9. organiser un programme EV3 de manière plus claire

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 2 roues identiques
4. 1 ordinateur avec Visual Studio Code
5. MicroPython installé sur l'EV3
6. un espace libre pour tester les déplacements

1. Rappel du cours précédent

Lors du cours précédent, nous avons utilisé `DriveBase`.

Par exemple :

```
robot.straight(500)
```

pour avancer de 500 mm.

Et :

```
robot.turn(90)
```

pour tourner de 90 degrés.

Nous pouvons déjà créer des parcours assez complexes.

Par exemple :

```
robot.straight(500)
robot.turn(90)

robot.straight(500)
robot.turn(90)

robot.straight(500)
robot.turn(90)
```

Mais plus le programme devient grand, plus il devient difficile à lire.

Nous allons maintenant apprendre à créer nos propres commandes.

2. Qu'est-ce qu'une fonction ?

Une fonction est un ensemble d'instructions auquel nous donnons un nom.

Par exemple, nous pouvons créer une fonction appelée :

```
avancer
```

Elle contiendra :

```
robot.straight(500)
```

Nous pourrons ensuite simplement écrire :

```
avancer()
```

On peut donc imaginer une fonction comme une nouvelle commande que nous ajoutons nous mêmes au langage.

3. Créer une première fonction

La syntaxe est :

```
def nom_de_la_fonction():
    instruction
```

Exemple :

```
def dire_bonjour():
    print("Bonjour")
```

Pour exécuter cette fonction, nous devons ensuite l'appeler :

```
dire_bonjour()
```

Programme complet :

```
def dire_bonjour():
    print("Bonjour")

dire_bonjour()
```

Résultat :

```
Bonjour
```

4. Comprendre `def`

Le mot :

```
def
```

permet de définir une fonction.

Dans :

```
def dire_bonjour():
```

nous créons une fonction appelée :

```
dire_bonjour
```

Les parenthèses font partie de la syntaxe.

Le symbole :

```
:
```

indique le début du contenu de la fonction.

Les instructions appartenant à la fonction sont indentées.

5. Définir n’est pas exécuter

Regardons :

```
def dire_bonjour():
    print("Bonjour")
```

Si le programme contient uniquement cela, rien n’est affiché.

Pourquoi ?

Parce que nous avons seulement créé la fonction.

Nous ne l’avons pas encore appelée.

Pour l’exécuter :

```
dire_bonjour()
```

6. Appeler plusieurs fois une fonction

```
def dire_bonjour():
    print("Bonjour")

dire_bonjour()
dire_bonjour()
dire_bonjour()
```

Résultat :

```
Bonjour
Bonjour
Bonjour
```

Une fonction peut donc être utilisée autant de fois que nécessaire.

7. Premier exemple EV3

Créons une fonction qui fait produire un son à la brique.

```
from pybricks.hubs import EV3Brick

ev3 = EV3Brick()

def bip():
    ev3.speaker.beep()

bip()
```

Nous venons de créer notre propre commande :

```
bip()
```

8. Exercice 1

Créer une fonction appelée :

```
trois_bips
```

qui produit trois sons.

Une solution possible :

```
from pybricks.hubs import EV3Brick
from pybricks.tools import wait

ev3 = EV3Brick()

def trois_bips():

    for i in range(3):
        ev3.speaker.beep()
        wait(500)

trois_bips()
```

9. Une fonction peut contenir plusieurs instructions

Par exemple :

```
def sequence():

    print("Début")

    ev3.speaker.beep()

    wait(1000)

    print("Fin")
```

Lorsque nous écrivons :

```
sequence()
```

toutes les instructions contenues dans la fonction sont exécutées.

10. Première fonction pour le robot

Nous pouvons créer :

```
def avancer():
    robot.straight(500)
```

Puis :

```
avancer()
```

Programme complet :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

def avancer():
    robot.straight(500)

avancer()
```

11. Créer une fonction pour tourner

```
def tourner_droite():
    robot.turn(90)
```

Nous pouvons maintenant écrire :

```
avancer()

tourner_droite()

avancer()
```

Le programme devient plus facile à comprendre.

12. Plusieurs fonctions

Nous pouvons créer :

```
def avancer():
    robot.straight(500)

def reculer():
    robot.straight(-500)

def tourner_droite():
    robot.turn(90)

def tourner_gauche():
    robot.turn(-90)
```

Nous avons maintenant créé quatre nouvelles commandes.

13. Utiliser les fonctions

```
avancer()

tourner_droite()

avancer()

tourner_gauche()

reculer()
```

Même quelqu'un qui connaît peu Python peut comprendre la logique du programme.

C'est l'un des grands avantages des fonctions.

14. Pourquoi utiliser des fonctions ?

Les fonctions permettent principalement de :

1. éviter les répétitions
2. rendre le programme plus lisible
3. organiser le code
4. réutiliser les mêmes instructions
5. modifier facilement un comportement

15. Exemple sans fonction

```
robot.straight(500)
robot.turn(90)

robot.straight(500)
robot.turn(90)

robot.straight(500)
robot.turn(90)

robot.straight(500)
robot.turn(90)
```

16. Avec une fonction

```
def cote_carre():
    robot.straight(500)
    robot.turn(90)
```

Puis :

```
for i in range(4):

    cote_carre()
```

Le programme est beaucoup plus compact.

17. Exercice pratique 2

Créer une fonction appelée :

```
demi_tour
```

qui fait tourner le robot de 180 degrés.

Solution :

```
def demi_tour():
    robot.turn(180)
```

Puis tester :

```
robot.straight(500)

demi_tour()

robot.straight(500)
```

Le robot devrait revenir approximativement vers son point de départ.

18. Le problème des fonctions trop fixes

Notre fonction :

```
def avancer():
    robot.straight(500)
```

avance toujours de 500 mm.

Mais peut être voulons nous parfois avancer de :

```
200 mm
```

ou :

```
800 mm
```

ou :

```
1000 mm
```

Créer une fonction différente pour chaque distance serait peu pratique.

Nous allons utiliser des paramètres.

19. Qu'est-ce qu'un paramètre ?

Un paramètre permet de transmettre une information à une fonction.

Exemple :

```
def avancer(distance):
    robot.straight(distance)
```

Nous pouvons maintenant écrire :

```
avancer(500)
```

ou :

```
avancer(1000)
```

ou :

```
avancer(250)
```

La même fonction peut donc effectuer plusieurs déplacements.

20. Comprendre le paramètre

Dans :

```
def avancer(distance):
```

`distance` est le paramètre.

Lorsque nous écrivons :

```
avancer(500)
```

la variable `distance` reçoit :

```
500
```

La fonction exécute donc :

```
robot.straight(500)
```

Si nous écrivons :

```
avancer(800)
```

elle exécute :

```
robot.straight(800)
```

21. Exercice 3

Créer une fonction :

```
def tourner(angle):
```

qui permet de faire tourner le robot.

Solution :

```
def tourner(angle):
    robot.turn(angle)
```

Nous pouvons ensuite écrire :

```
tourner(90)

tourner(-90)

tourner(180)
```

22. Créer des commandes simples

Nous pouvons maintenant avoir :

```
def avancer(distance):
    robot.straight(distance)

def tourner(angle):
    robot.turn(angle)
```

Puis construire un parcours :

```
avancer(500)

tourner(90)

avancer(300)

tourner(-90)

avancer(800)
```

Le programme est beaucoup plus proche d'une description naturelle du trajet.

23. Fonction avec plusieurs paramètres

Une fonction peut recevoir plusieurs informations.

Exemple :

```
def mouvement(distance, angle):
    robot.straight(distance)
    robot.turn(angle)
```

Puis :

```
mouvement(500, 90)
```

La fonction effectue :

```
avancer de 500 mm

puis

tourner de 90 degrés
```

24. Exercice pratique 4

Créer une fonction :

```
def cote(distance, angle):
```

qui :

```
avance d'une certaine distance

puis

tourne d'un certain angle
```

Solution :

```
def cote(distance, angle):
    robot.straight(distance)
    robot.turn(angle)
```

Puis utiliser :

```
for i in range(4):
    cote(500, 90)
```

25. Créer une fonction `carre()`

Nous pouvons maintenant aller plus loin.

```
def carre():

    for i in range(4):

        robot.straight(500)

        robot.turn(90)
```

Puis simplement écrire :

```
carre()
```

Le robot réalise le carré.

26. Fonction avec paramètre

Encore mieux :

```
def carre(taille):

    for i in range(4):

        robot.straight(taille)

        robot.turn(90)
```

Nous pouvons maintenant demander :

```
carre(200)
```

ou :

```
carre(500)
```

ou :

```
carre(1000)
```

27. Exercice pratique 5

Créer une fonction permettant de dessiner un triangle équilatéral.

Structure :

```
def triangle(taille):
```

Solution possible :

```
def triangle(taille):
    for i in range(3):
        robot.straight(taille)
        robot.turn(120)
```

Puis :

```
triangle(400)
```

28. Fonction générique pour un polygone

Nous pouvons réutiliser ce que nous avons appris au cours précédent.

```
def polygone(cotes, taille):
    angle = 360 / cotes
    for i in range(cotes):
        robot.straight(taille)
        robot.turn(angle)
```

Nous pouvons maintenant écrire :

```
polygone(3, 400)
```

pour un triangle.

```
polygone(4, 400)
```

pour un carré.

```
polygone(6, 300)
```

pour un hexagone.

29. Comprendre l'intérêt

Avec une seule fonction :

```
polygone(cotes, taille)
```

nous pouvons créer de nombreuses formes.

C'est un bon exemple de programme réutilisable.

30. Les paramètres comme variables locales

Un paramètre peut être utilisé comme une variable dans la fonction.

Exemple :

```
def double(nombre):
    resultat = nombre * 2
    print(resultat)
```

Puis :

```
double(5)
```

affiche :

```
10
```

31. Variable locale

Dans :

```
def double(nombre):
    resultat = nombre * 2
```

la variable :

```
resultat
```

est créée à l'intérieur de la fonction.

Elle est principalement destinée à être utilisée dans cette fonction.

On appelle cela une variable locale.

32. Retourner une valeur

Une fonction peut aussi calculer une information et la renvoyer.

Pour cela, on utilise :

```
return
```

Exemple :

```
def double(nombre):

    resultat = nombre * 2

    return resultat
```

Puis :

```
valeur = double(5)

print(valeur)
```

Résultat :

```
10
```

33. Différence entre `print()` et `return`

Prenons :

```
def double(nombre):
    print(nombre * 2)
```

Cette fonction affiche simplement la valeur.

Avec :

```
def double(nombre):
    return nombre * 2
```

la fonction renvoie la valeur au programme.

Nous pouvons ensuite la stocker :

```
resultat = double(5)
```

ou l'utiliser dans un autre calcul :

```
resultat = double(5) + 10
```

34. Exemple robotique avec `return`

Nous pouvons créer une fonction qui calcule un angle pour un polygone.

```
def calcul_angle(cotes):
```

```
    angle = 360 / cotes

    return angle
```

Puis :

```
angle = calcul_angle(4)

print(angle)
```

Résultat :

```
90
```

35. Utiliser directement la valeur retournée

```
def calcul_angle(cotes):
    return 360 / cotes
```

Puis :

```
robot.turn(calcul_angle(4))
```

Cela revient à demander :

```
robot.turn(90)
```

36. Exercice 6

Créer une fonction qui reçoit une distance en centimètres et retourne cette distance en millimètres.

Rappel :

```
1 cm = 10 mm
```

Solution :

```
def cm_vers_mm(distance_cm):

    distance_mm = distance_cm * 10

    return distance_mm
```

Puis :

```
distance = cm_vers_mm(50)

print(distance)
```

Résultat :

```
500
```

37. Application directe au robot

```
def cm_vers_mm(distance_cm):
    return distance_cm * 10
```

Puis :

```
robot.straight(cm_vers_mm(50))
```

Le robot avance de :

```
500 mm
```

soit :

```
50 cm
```

38. Créer une fonction `avancer_cm()`

Nous pouvons simplifier encore plus.

```
def avancer_cm(distance_cm):

    distance_mm = distance_cm * 10

    robot.straight(distance_mm)
```

Puis :

```
avancer_cm(50)
```

Cela est parfois plus naturel qu'utiliser directement les millimètres.

39. Fonction utilisant une condition

Une fonction peut contenir un `if`.

Par exemple :

```
def verifier_distance(distance):

    if distance < 200:
        print("Obstacle proche")
    else:
        print("Zone libre")
```

Puis :

```
verifier_distance(150)
```

Résultat :

```
Obstacle proche
```

40. Fonction utilisant une boucle

Une fonction peut également contenir une boucle.

Exemple :

```
def bips(nombre):

    for i in range(nombre):

        ev3.speaker.beep()

        wait(300)
```

Puis :

```
bips(5)
```

La brique produit cinq sons.

41. Une fonction peut appeler une autre fonction

Exemple :

```
def avancer(distance):
    robot.straight(distance)

def tourner(angle):
    robot.turn(angle)

def carre(taille):

    for i in range(4):

        avancer(taille)

        tourner(90)
```

Puis :

```
carre(500)
```

Nous avons maintenant une organisation en plusieurs niveaux.

42. Organisation du programme

Un programme bien organisé peut ressembler à ceci :

```
# Imports

from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


# Configuration

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


# Fonctions

def avancer(distance):
    robot.straight(distance)

def tourner(angle):
    robot.turn(angle)

def carre(taille):

    for i in range(4):

        avancer(taille)

        tourner(90)


# Programme principal

carre(500)
```

Cette structure est beaucoup plus facile à comprendre qu'un programme contenant toutes les instructions mélangées.

43. Le programme principal

Nous pouvons imaginer trois grandes parties :

```
IMPORTS
  ↓
CONFIGURATION
  ↓
FONCTIONS
  ↓
PROGRAMME PRINCIPAL
```

Cette organisation deviendra importante lorsque nos programmes feront plusieurs centaines de lignes.

44. Éviter les mauvais noms

Une fonction doit avoir un nom compréhensible.

Éviter :

```
def f1():
```

ou :

```
def x():
```

Préférer :

```
def avancer():
def tourner():
def detecter_obstacle():
def suivre_ligne():
```

Le nom doit expliquer ce que fait la fonction.

45. Une fonction doit idéalement avoir un rôle clair

Une fonction appelée :

```
def avancer():
```

devrait principalement s'occuper d'avancer.

Éviter une fonction qui :

```
avance

fait un son

affiche une image

attend 5 secondes

tourne

recalcule une variable
```

sans raison claire.

Une bonne fonction réalise généralement une tâche précise.

46. Exercice de lecture

Que fait ce programme ?

```
def trajet():

    robot.straight(500)

    robot.turn(90)

    robot.straight(300)
```

Puis :

```
trajet()
trajet()
```

Réponse :

Le robot effectue deux fois la même séquence.

47. Mini défi 1

Créer les fonctions suivantes :

```
avancer(distance)

reculer(distance)

tourner_droite()

tourner_gauche()
```

Solution possible :

```
def avancer(distance):
    robot.straight(distance)

def reculer(distance):
    robot.straight(-distance)

def tourner_droite():
    robot.turn(90)

def tourner_gauche():
    robot.turn(-90)
```

48. Mini défi 2

Utiliser uniquement ces fonctions pour créer :

```
Avancer de 500 mm

Tourner à droite

Avancer de 300 mm

Tourner à gauche

Reculer de 200 mm
```

Solution :

```
avancer(500)

tourner_droite()

avancer(300)

tourner_gauche()

reculer(200)
```

49. Mini défi 3

Créer :

```
def rectangle(longueur, largeur):
```

qui réalise un rectangle.

Solution possible :

```
def rectangle(longueur, largeur):

    for i in range(2):

        robot.straight(longueur)
        robot.turn(90)

        robot.straight(largeur)
        robot.turn(90)
```

Puis tester :

```
rectangle(600, 300)
```

50. Mini défi 4

Créer :

```
def polygone(cotes, taille):
```

qui fonctionne pour plusieurs formes.

Solution :

```
def polygone(cotes, taille):
    angle = 360 / cotes
    for i in range(cotes):
        robot.straight(taille)
        robot.turn(angle)
```

Tester :

```
polygone(3, 300)
polygone(4, 300)
polygone(5, 300)
```

51. Défi final : créer une bibliothèque de mouvements

L'objectif est de créer un programme disposant de plusieurs commandes réutilisables.

Tu dois créer au minimum :

```
avancer(distance)
reculer(distance)
tourner(angle)
carre(taille)
rectangle(longueur, largeur)
polygone(cotes, taille)
```

Puis réaliser un parcours uniquement en utilisant leurs fonctions.

Par exemple :

```
Carré de 300 mm
↓
Avancer de 500 mm
↓
Tourner de 180°
↓
Rectangle de 600 sur 300 mm
↓
Reculer de 200 mm
```

52. Exemple de solution complète

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


def avancer(distance):

    robot.straight(distance)


def reculer(distance):

    robot.straight(-distance)


def tourner(angle):

    robot.turn(angle)


def carre(taille):

    for i in range(4):

        avancer(taille)

        tourner(90)


def rectangle(longueur, largeur):

    for i in range(2):

        avancer(longueur)

        tourner(90)

        avancer(largeur)

        tourner(90)


def polygone(cotes, taille):

    angle = 360 / cotes

    for i in range(cotes):

        avancer(taille)

        tourner(angle)
```

```
carre(300)

avancer(500)

tourner(180)

rectangle(600, 300)

reculer(200)
```

53. Questions de fin de cours

Question 1

À quoi sert une fonction ?

Question 2

Quel mot permet de créer une fonction en Python ?

Question 3

Quelle différence existe entre :

```
def avancer():
```

et :

```
avancer()
```

Question 4

À quoi sert un paramètre ?

Question 5

Que signifie :

```
def avancer(distance):
```

Question 6

Une fonction peut elle avoir plusieurs paramètres ?

Question 7

À quoi sert `return` ?

Question 8

Pourquoi les fonctions rendent elles un programme plus facile à maintenir ?

Correction

Réponse 1

Une fonction permet de regrouper plusieurs instructions sous un même nom afin de pouvoir les réutiliser.

Réponse 2

`def`

Réponse 3

`def avancer():`

crée la fonction.

`avancer()`

exécute la fonction.

Réponse 4

Un paramètre permet de transmettre une valeur à la fonction.

Réponse 5

La fonction `avancer` reçoit une valeur appelée `distance`.

Réponse 6

Oui.

Par exemple :

`def rectangle(longueur, largeur):`

possède deux paramètres.

Réponse 7

`return` permet à une fonction de renvoyer une valeur.

Réponse 8

Parce qu'une modification peut être effectuée une seule fois dans la fonction au lieu de modifier toutes les occurrences du même code.

Difficultés fréquentes

Erreur 1 : oublier les parenthèses

Incorrect :

```
avancer
```

Correct :

```
avancer()
```

Erreur 2 : appeler une fonction avec le mauvais nombre de paramètres

Si nous avons :

```
def avancer(distance):
```

ceci est incorrect :

```
avancer()
```

Il manque la distance.

Il faut par exemple écrire :

```
avancer(500)
```

Erreur 3 : oublier l'indentation

Incorrect :

```
def avancer(distance):
robot.straight(distance)
```

Correct :

```
def avancer(distance):
    robot.straight(distance)
```

Erreur 4 : confondre `print()` et `return`

```
print(resultat)
```

affiche une valeur.

```
return resultat
```

renvoie la valeur à l'endroit où la fonction a été appelée.

Cours 11 : Paramètres, `return` et fonctions réutilisables avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. utiliser plusieurs paramètres dans une fonction
2. distinguer paramètre et argument
3. utiliser `return`
4. récupérer une valeur retournée par une fonction
5. utiliser une fonction dans un calcul
6. créer des fonctions de conversion
7. créer des fonctions de calcul utiles au robot
8. rendre un programme EV3 plus facile à modifier
9. découper un problème complexe en plusieurs petites fonctions

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 moteurs EV3
3. 2 roues identiques
4. 1 ordinateur avec Visual Studio Code
5. MicroPython installé sur l'EV3
6. une règle ou un mètre
7. un espace permettant de tester le robot

1. Rappel du cours précédent

Lors du cours précédent, nous avons appris à créer des fonctions.

Par exemple :

```
def avancer(distance):
    robot.straight(distance)
```

Nous pouvons ensuite écrire :

```
avancer(500)
```

Le robot avance alors de :

```
500 mm
```

Nous avons également créé :

```
def tourner(angle):
    robot.turn(angle)
```

Puis :

```
tourner(90)
```

Aujourd'hui, nous allons approfondir le fonctionnement des fonctions.

2. Paramètre et argument

Ces deux mots sont souvent utilisés en programmation.

Prenons :

```
def avancer(distance):
    robot.straight(distance)
```

Dans cette définition :

```
distance
```

est un **paramètre**.

Lorsque nous appelons :

```
avancer(500)
```

la valeur :

```
500
```

est un **argument**.

On peut donc retenir :

```
PARAMÈTRE

nom utilisé dans la définition
```

et :

```
ARGUMENT

valeur envoyée lors de l'appel
```

3. Exemple simple

```
def afficher_nombre(nombre):
    print(nombre)

afficher_nombre(10)
```

Ici :

```
nombre
```

est le paramètre.

Et :

```
10
```

est l'argument.

4. Plusieurs paramètres

Une fonction peut recevoir plusieurs paramètres.

Par exemple :

```
def addition(a, b):
    resultat = a + b
    print(resultat)
```

Nous pouvons écrire :

```
addition(10, 5)
```

Le programme affiche :

```
15
```

5. Application au robot

Créons une fonction :

```
def deplacement(distance, angle):
    robot.straight(distance)
    robot.turn(angle)
```

Puis :

```
deplacement(500, 90)
```

Le robot :

```
avance de 500 mm

puis

tourne de 90°
```

6. Exercice 1

Créer une fonction :

```
def aller_retour(distance):
```

qui fait :

```
Avancer de la distance demandée

Puis

Reculer de la même distance
```

Solution :

```
def aller_retour(distance):
    robot.straight(distance)
    robot.straight(-distance)
```

Puis tester :

```
aller_retour(500)
```

7. Plusieurs arguments

Prenons :

```
def rectangle(longueur, largeur):
```

Nous devons fournir deux valeurs.

Par exemple :

```
rectangle(600, 300)
```

Ici :

```
longueur = 600

largeur = 300
```

8. L’ordre des arguments est important

Si nous avons :

```
def rectangle(longueur, largeur):
```

et que nous écrivons :

```
rectangle(600, 300)
```

Python associe :

```
longueur = 600

largeur = 300
```

Mais si nous écrivons :

```
rectangle(300, 600)
```

nous obtenons :

```
longueur = 300

largeur = 600
```

L’ordre des valeurs est donc important.

9. Qu’est-ce que `return` ?

Certaines fonctions ne réalisent pas directement une action.

Elles calculent une valeur.

Exemple :

```
def double(nombre):
    resultat = nombre * 2
    return resultat
```

Puis :

```
valeur = double(5)

print(valeur)
```

Résultat :

```
10
```

10. Comprendre `return`

On peut imaginer :

```
Entrée

5
↓
Fonction double()
↓
Calcul

5 × 2
↓
RETURN
↓
10
```

La fonction renvoie donc une information au programme principal.

11. Fonction sans `return`

```
def double(nombre):
    print(nombre * 2)
```

Cette fonction affiche le résultat.

Mais nous ne pouvons pas facilement réutiliser cette valeur.

Par exemple :

```
resultat = double(5)
```

ne permet pas de récupérer `10` comme avec `return`.

12. Fonction avec `return`

```
def double(nombre):
    return nombre * 2
```

Nous pouvons maintenant écrire :

```
resultat = double(5)
```

La variable :

```
resultat
```

contient :

```
10
```

13. Utiliser directement une valeur retournée

Nous pouvons écrire :

```
print(double(5))
```

ou même :

```
nouveau_nombre = double(5) + 10
```

Calcul :

```
double(5)
↓
10
↓
10 + 10
↓
20
```

14. Conversion centimètres vers millimètres

`DriveBase` utilise des millimètres pour `straight()`.

Mais tu peux préférer parfois travailler en centimètres.

Créons donc :

```
def cm_vers_mm(distance_cm):
    return distance_cm * 10
```

Puis :

```
distance = cm_vers_mm(50)

print(distance)
```

Résultat :

```
500
```

15. Application directe au robot

```
robot.straight(cm_vers_mm(50))
```

La fonction retourne :

```
500
```

Le robot reçoit donc en réalité :

```
robot.straight(500)
```

16. Créer une fonction `avancer_cm()`

Nous pouvons également cacher complètement la conversion.

```
def avancer_cm(distance_cm):
    distance_mm = distance_cm * 10
    robot.straight(distance_mm)
```

Puis :

```
avancer_cm(50)
```

Le robot avance de :

```
50 cm
```

17. Exercice 2

Créer :

```
def reculer_cm(distance_cm):
```

Solution :

```
def reculer_cm(distance_cm):
    distance_mm = distance_cm * 10
    robot.straight(-distance_mm)
```

Tester :

```
reculer_cm(30)
```

18. Calculer l'angle d'un polygone

Nous avons vu que l'angle extérieur d'un polygone régulier est :

```
360 / nombre de côtés
```

Nous pouvons créer :

```
def calcul_angle(cotes):
    return 360 / cotes
```

Puis :

```
angle = calcul_angle(4)

print(angle)
```

Résultat :

```
90
```

19. Utiliser cette fonction dans une autre fonction

```
def calcul_angle(cotes):
    return 360 / cotes

def polygone(cotes, taille):

    angle = calcul_angle(cotes)

    for i in range(cotes):
        robot.straight(taille)
        robot.turn(angle)
```

Puis :

```
polygone(6, 300)
```

La première fonction calcule.
La deuxième utilise le résultat.

20. Pourquoi découper le programme ?

Nous pourrions écrire directement :

```
angle = 360 / cotes
```
dans `polygone()`.

Mais séparer les calculs devient intéressant lorsque :

```
le calcul est complexe

le même calcul est utilisé plusieurs fois
nous voulons tester le calcul séparément
nous voulons rendre le code plus lisible
```

21. Une fonction pour calculer le périmètre d'une roue

Rappel :

```
Périmètre = π × diamètre
```

Créons :

```
def perimetre_roue(diametre):
    return 3.14 * diametre
```

Puis :

```
perimetre = perimetre_roue(56)

print(perimetre)
```

Résultat approximatif :

```
175,84 mm
```

22. Calculer le nombre de rotations

Nous pouvons créer :

```
def calcul_rotations(distance, diametre):
    perimetre = 3.14 * diametre
    rotations = distance / perimetre
    return rotations
```

Puis :

```
rotations = calcul_rotations(1000, 56)
print(rotations)
```

Résultat approximatif :

```
5,69
```

23. Calculer l'angle moteur

Une rotation vaut :

```
360 degrés
```

Nous pouvons donc créer :

```
def calcul_angle_moteur(distance, diametre):
    perimetre = 3.14 * diametre
    rotations = distance / perimetre
    angle = rotations * 360
    return angle
```

Puis :

```
angle = calcul_angle_moteur(1000, 56)
print(angle)
```

Résultat approximatif :

```
2048 degrés
```

24. Exercice 3

Créer une fonction :

```
def secondes_vers_ms(secondes):
```

qui retourne une durée en millisecondes.

Solution :

```
def secondes_vers_ms(secondes):
    return secondes * 1000
```

Test :

```
duree = secondes_vers_ms(3)

print(duree)
```

Résultat :

```
3000
```

25. Plusieurs fonctions peuvent travailler ensemble

Exemple :

```
def cm_vers_mm(cm):
    return cm * 10


def avancer_cm(cm):
    distance = cm_vers_mm(cm)
    robot.straight(distance)
```

Puis :

```
avancer_cm(75)
```

La première fonction effectue la conversion.

La deuxième déplace le robot.

26. Fonctions et conditions

Une fonction peut prendre une décision.

Exemple :

```
def choisir_vitesse(distance):

    if distance < 200:
        return 0

    elif distance < 500:
        return 100

    else:
        return 300
```

Puis :

```
vitesse = choisir_vitesse(350)

print(vitesse)
```

Résultat :

```
100
```

27. Comprendre ce programme

La fonction reçoit une distance.

Elle décide ensuite quelle vitesse utiliser.

```
distance < 200

↓

0
distance entre 200 et 499

↓

100
distance égale ou supérieure à 500

↓

300
```

Cette logique deviendra très utile avec le capteur ultrason.

28. `return` peut arrêter une fonction

Prenons :

```
def verifier_nombre(nombre):
```

```
    if nombre < 0:
        return "Nombre négatif"

    return "Nombre positif ou nul"
```

Si :

```
nombre < 0
```

Python exécute immédiatement :

```
return "Nombre négatif"
```

et quitte la fonction.

29. Exemple avec une sécurité

Imaginons :

```
def verifier_vitesse(vitesse):

    if vitesse > 1000:
        return 1000

    return vitesse
```

Puis :

```
vitesse = verifier_vitesse(1500)

print(vitesse)
```

Résultat :

```
1000
```

La fonction empêche donc une valeur supérieure à la limite choisie.

30. Limiter une valeur

Nous pouvons aussi vérifier les valeurs négatives.

```
def limiter_vitesse(vitesse):

    if vitesse > 1000:
        return 1000

    if vitesse < 0:
        return 0

    return vitesse
```

Exemples :

```
print(limiter_vitesse(500))
```

donne :

```
500
print(limiter_vitesse(1500))
```

donne :

```
1000
print(limiter_vitesse(-100))
```

donne :

```
0
```

31. Exercice 4

Créer une fonction qui limite un angle entre :

```
moins 180 degrés

et

180 degrés
```

Une solution possible :

```
def limiter_angle(angle):

    if angle > 180:
        return 180

    if angle < -180:
        return -180

    return angle
```

32. Paramètres avec valeurs par défaut

Python permet aussi de donner une valeur par défaut à un paramètre.

Exemple :

```
def avancer(distance=500):
    robot.straight(distance)
```

Nous pouvons appeler :

```
avancer()
```

La fonction utilise alors :

```
500
```

Mais nous pouvons aussi écrire :

```
avancer(1000)
```

et la valeur fournie remplace la valeur par défaut.

33. Exemple avec un angle par défaut

```
def tourner(angle=90):
    robot.turn(angle)
```

Puis :

```
tourner()
```

effectue :

```
90 degrés
```

Alors que :

```
tourner(180)
```

effectue :

```
180 degrés
```

34. Pourquoi utiliser une valeur par défaut ?

Cela peut être pratique lorsqu'une valeur est utilisée très souvent.

Par exemple, si le robot tourne presque toujours de :

```
90 degrés
```

nous pouvons écrire :

```
def tourner(angle=90):
    robot.turn(angle)
```

Puis :

```
tourner()
```

dans la majorité des cas.

35. Exercice 5

Créer :

```
def bip(nombre=1):
```

qui produit un ou plusieurs sons.

Solution :

```
def bip(nombre=1):

    for i in range(nombre):
        ev3.speaker.beep()
        wait(300)
```

Puis :

```
bip()
```

fait un son.

Et :

```
bip(5)
```

fait cinq sons.

36. Créer une fonction de trajet

Nous pouvons maintenant construire des fonctions plus élaborées.

```
def trajet(longueur, largeur):

    robot.straight(longueur)

    robot.turn(90)

    robot.straight(largeur)

    robot.turn(-90)
```

Puis :

```
trajet(500, 300)
```

37. Réutiliser la même fonction

```
trajet(500, 300)

trajet(700, 200)

trajet(1000, 500)
```

La logique reste la même.

Seules les données changent.

C’est l’un des principaux intérêts des paramètres.

38. Fonction pour un carré

Nous avions :

```
def carre(taille):
    for i in range(4):
        robot.straight(taille)
        robot.turn(90)
```

Nous pouvons améliorer cette fonction :

```
def carre(taille, sens=1):
    angle = 90 * sens
    for i in range(4):
        robot.straight(taille)
        robot.turn(angle)
```

39. Utiliser le sens

Avec :

```
carre(500, 1)
```

le robot tourne dans un sens.

Avec :

```
carre(500, -1)
```

il tourne dans l’autre sens.

Cela montre qu’un paramètre peut changer complètement le comportement d’une fonction.

40. Mini défi 1

Créer une fonction :

```
def triangle(taille, sens=1):
```

Une solution possible :

```
def triangle(taille, sens=1):
    angle = 120 * sens
    for i in range(3):
        robot.straight(taille)
        robot.turn(angle)
```

41. Mini défi 2

Créer :

```
def polygone(cotes, taille, sens=1):
```

Solution possible :

```
def polygone(cotes, taille, sens=1):
    angle = (360 / cotes) * sens
    for i in range(cotes):
        robot.straight(taille)
        robot.turn(angle)
```

Tester :

```
polygone(5, 300, 1)
```

Puis :

```
polygone(5, 300, -1)
```

42. Fonctions de haut niveau

Nous pouvons commencer à créer plusieurs niveaux de fonctions.

Par exemple :

```
def avancer(distance):
    robot.straight(distance)


def tourner(angle):
    robot.turn(angle)


def cote(taille, angle):
    avancer(taille)
    tourner(angle)


def carre(taille):

    for i in range(4):
        cote(taille, 90)
```

Puis :

```
carre(500)
```

`carre()` utilise `cote()`.

`cote()` utilise `avancer()` et `tourner()`.

43. Visualiser les niveaux

```
carre()

↓

cote()

↓

avancer()
tourner()

↓

DriveBase

↓

moteurs
```

Chaque niveau cache une partie de la complexité.

C'est ce qu'on appelle l'abstraction.

44. Pourquoi l’abstraction est elle utile ?

Imagine un programme de :

```
500 lignes
```

Si chaque déplacement est écrit directement avec les moteurs, le programme devient difficile à comprendre.

Avec des fonctions comme :

```
chercher_ligne()
eviter_obstacle()
retourner_base()
```

le programme principal peut rester très lisible.

45. Exemple de programme principal

Un jour, nous pourrons écrire quelque chose comme :

```
avancer_jusqua_ligne()

tourner(90)

suivre_ligne()

eviter_obstacle()

retourner_base()
```

Chaque fonction pourra contenir plusieurs dizaines de lignes.

Mais le comportement général du robot restera facile à lire.

46. Exercice de conception

On veut programmer un robot qui :

```
avance de 50 cm

tourne de 90°

avance de 1 mètre

fait demi tour

revient de 1 mètre
```

Quelles fonctions pourrait on créer ?

Une réponse possible :

```
def avancer_cm(distance):
def tourner(angle):
def demi_tour():
```

Le programme principal pourrait devenir :

```
avancer_cm(50)

tourner(90)

avancer_cm(100)

demi_tour()

avancer_cm(100)
```

47. Mini projet : bibliothèque de déplacement

Créer les fonctions suivantes :

```
avancer_mm(distance)
avancer_cm(distance)
reculer_mm(distance)
reculer_cm(distance)
tourner(angle)
demi_tour()
carre(taille)
rectangle(longueur, largeur)
polygone(cotes, taille)
```

48. Exemple de solution

```
def avancer_mm(distance):
    robot.straight(distance)


def avancer_cm(distance):
    robot.straight(distance * 10)


def reculer_mm(distance):
    robot.straight(-distance)


def reculer_cm(distance):
    robot.straight(-(distance * 10))


def tourner(angle):
    robot.turn(angle)


def demi_tour():
    robot.turn(180)


def carre(taille):

    for i in range(4):
        avancer_mm(taille)
        tourner(90)


def rectangle(longueur, largeur):

    for i in range(2):

        avancer_mm(longueur)
        tourner(90)

        avancer_mm(largeur)
        tourner(90)


def polygone(cotes, taille):

    angle = 360 / cotes

    for i in range(cotes):

        avancer_mm(taille)

        tourner(angle)
```

49. Défi final

Créer une fonction :

```
def parcours(longueur, largeur, repetitions):
```

Le robot doit répéter :

```
Avancer de longueur
Tourner de 90°
Avancer de largeur
Tourner de moins 90°
```

le nombre de fois demandé.

Exemple

```
parcours(500, 300, 3)
```

doit produire trois motifs.

Solution possible

```
def parcours(longueur, largeur, repetitions):
    for i in range(repetitions):
        robot.straight(longueur)
        robot.turn(90)
        robot.straight(largeur)
        robot.turn(-90)
```

50. Défi supplémentaire : fonction avec résultat

Créer :

```
def distance_totale(longueur, largeur, repetitions):
```

qui calcule la distance totale parcourue par le robot.

Pour une répétition :

```
distance = longueur + largeur
```

Donc :

```
def distance_totale(longueur, largeur, repetitions):
    distance = (longueur + largeur) * repetitions
    return distance
```

Puis :

```
total = distance_totale(500, 300, 3)

print(total)
```

Résultat :

```
2400 mm
```

51. Programme complet d'exemple

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


def cm_vers_mm(cm):
    return cm * 10


def avancer_cm(cm):
    robot.straight(cm_vers_mm(cm))


def tourner(angle=90):
    robot.turn(angle)


def calcul_angle_polygone(cotes):
    return 360 / cotes


def polygone(cotes, taille_cm):

    angle = calcul_angle_polygone(cotes)

    distance = cm_vers_mm(taille_cm)

    for i in range(cotes):

        robot.straight(distance)

        robot.turn(angle)


avancer_cm(50)

tourner()

polygone(5, 30)
```

52. Questions de fin de cours

Question 1

Quelle différence existe entre un paramètre et un argument ?

Question 2

À quoi sert `return` ?

Question 3

Que retourne :

```
def double(nombre):
    return nombre * 2
```

si on appelle :

```
double(10)
```

Question 4

Combien de paramètres possède :

```
def rectangle(longueur, largeur):
```

Question 5

Que signifie :

```
def tourner(angle=90):
```

Question 6

Quelle valeur sera utilisée avec :

```
tourner()
```

Question 7

Quelle valeur sera utilisée avec :

```
tourner(180)
```

Question 8

Pourquoi créer plusieurs petites fonctions plutôt qu'une seule énorme fonction ?

Correction

Réponse 1

Le paramètre est le nom utilisé dans la définition de la fonction.

L'argument est la valeur envoyée lorsqu'on appelle la fonction.

Réponse 2

`return` permet de renvoyer une valeur vers le programme qui a appelé la fonction.

Réponse 3

`20`

Réponse 4

Deux paramètres :

`longueur`

`largeur`

Réponse 5

Le paramètre `angle` possède une valeur par défaut de :

`90`

Réponse 6

`90°`

Réponse 7

`180°`

Réponse 8

Parce que plusieurs petites fonctions sont généralement plus faciles à comprendre, tester, réutiliser et modifier.

Erreurs fréquentes

Erreur 1 : oublier `return`

Si tu écris :

```
def calcul_angle(cotes):
    angle = 360 / cotes
```

Puis :
```
angle = calcul_angle(4)
```

La fonction calcule bien une valeur, mais ne la renvoie pas.

Correct :
```
def calcul_angle(cotes):
    angle = 360 / cotes
    return angle
```

Erreur 2 : mauvais nombre d'arguments

Fonction :

```
def rectangle(longueur, largeur):
```

Appel incorrect :
```
rectangle(500)
```

Il manque une valeur.

Appel correct :
```
rectangle(500, 300)
```

Erreur 3 : mauvais ordre

```
rectangle(300, 600)
```

ne donne pas les mêmes paramètres que :

```
rectangle(600, 300)
```

Erreur 4 : utiliser une fonction avant de l'avoir définie

Il est préférable de placer les définitions avant le programme principal.

Structure recommandée :

```
Imports

Configuration

Fonctions

Programme principal
```

Cours 12 : Projet de synthèse avant les capteurs

Objectifs du cours

Cette leçon sert à consolider toutes les notions vues jusqu'à présent.

À la fin de cette leçon, tu seras capable de :

1. analyser un parcours
2. découper un problème en plusieurs étapes
3. utiliser des variables
4. effectuer des calculs simples
5. utiliser `if`
6. utiliser `for`
7. utiliser `while`
8. utiliser `DriveBase`
9. créer des fonctions
10. utiliser des paramètres
11. utiliser `return`
12. organiser correctement un programme
13. tester et corriger un robot réel

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3 par étudiant
2. 2 grands moteurs EV3
3. 2 roues identiques
4. 1 roulette ou point d'appui
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. un mètre
8. du ruban adhésif
9. plusieurs objets servant de repères
10. un espace libre au sol

1. Introduction

Depuis le début du cours, nous avons appris de nombreuses notions.

Python

`variables`

`calculs`

`conditions`

`boucles while`

`boucles for`

`fonctions`

`paramètres`

`return`

Robotique

`moteurs`

`vitesse`

`angle`

`distance`

`deux moteurs`

`DriveBase`

`calibration`

`trajectoires`

Aujourd'hui, nous allons réunir toutes ces notions dans un seul projet.

2. Le défi

Le robot doit suivre un parcours imposé.

Par exemple :

```
DÉPART
↓
Avancer de 60 cm
↓
Tourner à droite de 90°
↓
Avancer de 40 cm
↓
Faire un demi tour
↓
Revenir de 40 cm
↓
Tourner à gauche de 90°
↓
Avancer de 30 cm
↓
Faire un carré de 25 cm de côté
↓
STOP
```

Le robot doit effectuer le parcours de manière aussi précise que possible.

3. Première étape : analyser le problème

Avant de programmer, il faut identifier les différentes actions nécessaires.

Nous avons :

```
1. avancer

2. tourner

3. avancer

4. faire demi tour

5. avancer

6. tourner

7. avancer

8. faire un carré
```

Nous pouvons maintenant réfléchir aux fonctions dont nous avons besoin.

4. Identifier les fonctions utiles

Nous pourrions créer :

```
def avancer_cm(distance):
def tourner(angle):
def demi_tour():
def carre(taille):
```

Notre programme principal deviendra alors beaucoup plus simple.

5. Initialisation du robot

Programme de départ :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

Attention : les valeurs de `wheel_diameter` et `axle_track` doivent correspondre au robot utilisé.

6. Première fonction : convertir les centimètres

`DriveBase` travaille en millimètres.

Nous voulons pouvoir écrire :

```
avancer_cm(60)
```

Créons donc :

```
def cm_vers_mm(distance_cm):
    return distance_cm * 10
```

7. Fonction `avancer_cm()`

```
def avancer_cm(distance_cm):

    distance_mm = cm_vers_mm(distance_cm)

    robot.straight(distance_mm)
```

Nous pouvons maintenant écrire :

```
avancer_cm(60)
```

au lieu de :

```
robot.straight(600)
```

8. Fonction pour tourner

```
def tourner(angle):
    robot.turn(angle)
```

Nous pouvons maintenant écrire :

```
tourner(90)
```

ou :

```
tourner(-90)
```

9. Fonction demi-tour

```
def demi_tour():
    robot.turn(180)
```

Le programme devient très lisible :

```
demi_tour()
```

10. Fonction pour un carré

```
def carre(taille_cm):

    taille_mm = cm_vers_mm(taille_cm)

    for i in range(4):

        robot.straight(taille_mm)

        robot.turn(90)
```

Nous pouvons maintenant écrire :

```
carre(25)
```

11. Programme principal

Notre parcours peut maintenant être décrit presque comme une série d'instructions naturelles.

```
avancer_cm(60)

tourner(90)

avancer_cm(40)

demi_tour()

avancer_cm(40)

tourner(-90)

avancer_cm(30)

carre(25)
```

12. Programme complet

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


def cm_vers_mm(distance_cm):

    return distance_cm * 10

def avancer_cm(distance_cm):

    distance_mm = cm_vers_mm(distance_cm)

    robot.straight(distance_mm)

def tourner(angle):

    robot.turn(angle)

def demi_tour():

    robot.turn(180)
```

```
def carre(taille_cm):

    taille_mm = cm_vers_mm(taille_cm)

    for i in range(4):

        robot.straight(taille_mm)

        robot.turn(90)

avancer_cm(60)

tourner(90)

avancer_cm(40)

demi_tour()

avancer_cm(40)

tourner(-90)

avancer_cm(30)

carre(25)
```

13. Pourquoi ce programme est-il bien organisé ?

Nous pouvons distinguer trois grandes parties.

Configuration

```
moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)
```

et :

```
robot = DriveBase(...)
```

Fonctions

```
cm_vers_mm()
avancer_cm()
tourner()
demi_tour()
carre()
```

Programme principal

```
avancer_cm(60)

tourner(90)

avancer_cm(40)
```

Cette organisation rend le programme beaucoup plus facile à comprendre.

14. Phase de test

Le programme ne doit pas être considéré comme terminé simplement parce qu'il fonctionne sans erreur Python.

Il faut maintenant tester le robot réel.

Le processus doit être :

```
PROGRAMMER
↓
TESTER
↓
MESURER
↓
IDENTIFIER L'ERREUR
↓
CORRIGER
↓
TESTER À NOUVEAU
```

15. Premier test : distance

Demander :

```
avancer_cm(100)
```

Le robot doit parcourir environ :

```
1 mètre
```

Faire trois essais.

Essai	Distance demandée	Distance réelle	Erreur
1	100 cm		
2	100 cm		
3	100 cm		

16. Calculer l'erreur

Nous pouvons calculer :

```
Erreur = distance demandée moins distance réelle
```

Par exemple :

```
100 cm demandés
```

```
97 cm réellement parcourus
```

donne :

```
100 moins 97 = 3 cm
```

17. Créer une fonction de calcul d’erreur

Nous pouvons écrire :

```
def calcul_erreur(valeur_voulue, valeur_reelle):
    return valeur_voulue - valeur_reelle
```

Par exemple :

```
erreur = calcul_erreur(100, 97)
print(erreur)
```

Résultat :

```
3
```

18. Erreur positive et négative

Si :

```
distance voulue = 100 cm
distance réelle = 97 cm
```

alors :

```
erreur = 3
```

Le robot n’est pas allé assez loin.

Si :

```
distance voulue = 100 cm
distance réelle = 103 cm
```

alors :

```
erreur = moins 3
```

Le robot est allé trop loin.

Cette notion d’erreur sera extrêmement importante plus tard.

19. Test des rotations

Nous devons également vérifier :

```
robot.turn(90)
```

Tracer au sol deux lignes formant un angle droit.

Faire plusieurs essais.

Essai	Angle demandé	Angle observé
1	90°	
2	90°	
3	90°	

20. Test de 360 degrés

Un bon moyen de calibrer les rotations consiste à utiliser :

```
robot.turn(360)
```

Le robot devrait revenir dans son orientation initiale.

On peut tracer une ligne devant lui avant le départ.

Après la rotation, vérifier son orientation.

21. Réduire l’erreur cumulative

Un petit écart sur chaque rotation peut devenir important après plusieurs rotations.

Par exemple :

```
89°
```

```
89°
```

```
89°
```

```
89°
```

donne :

```
356°
```

au lieu de :

```
360°
```

L’erreur totale est donc :

```
4°
```

22. Défi 1 : le carré précis

Créer un carré de :

```
50 cm de côté
```

Programme :

```
for i in range(4):
    avancer_cm(50)
    tourner(90)
```

Objectif :

Le robot doit revenir aussi proche que possible de :

```
sa position de départ
et
son orientation de départ
```

23. Mesurer le résultat

Après le carré, mesurer :

```
distance entre position de départ et position finale
```

Noter :

Essai	Erreur de position
1	
2	
3	

L'étudiant ayant l'erreur moyenne la plus faible peut recevoir un point supplémentaire.

24. Défi 2 : rectangle

Créer une fonction :

```
def rectangle(longueur_cm, largeur_cm):
```

Solution possible :

```
def rectangle(longueur_cm, largeur_cm):
    for i in range(2):
        avancer_cm(longueur_cm)
        tourner(90)
        avancer_cm(largeur_cm)
        tourner(90)
```

Tester :

```
rectangle(80, 40)
```

25. Défi 3 : polygone

Créer :

```
def polygone(cotes, taille_cm):
```

Solution :

```
def polygone(cotes, taille_cm):
    angle = 360 / cotes
    for i in range(cotes):
        avancer_cm(taille_cm)
        tourner(angle)
```

Tester :

```
polygone(3, 30)
polygone(4, 30)
polygone(5, 30)
polygone(6, 30)
```

26. Utiliser une condition

Nous pouvons améliorer la fonction.

Que se passe t-il si quelqu'un écrit :

```
polygone(2, 30)
```

Un polygone doit posséder au minimum :

```
3 côtés
```

Nous pouvons donc vérifier cette valeur.

```
def polygone(cotes, taille_cm):

    if cotes < 3:
        print("Erreur : minimum 3 côtés")
        return

    angle = 360 / cotes

    for i in range(cotes):

        avancer_cm(taille_cm)

        tourner(angle)
```

27. Pourquoi utiliser `return` ici ?

Dans :

```
if cotes < 3:
    print("Erreur")
    return
```

`return` arrête immédiatement la fonction.

Le programme ne continue donc pas avec un nombre de côtés incorrect.

28. Exercice

Ajouter également une vérification pour empêcher :

```
taille inférieure ou égale à zéro
```

Solution possible :

```
def polygone(cotes, taille_cm):

    if cotes < 3:
        print("Erreur : minimum 3 côtés")
        return

    if taille_cm <= 0:
        print("Erreur : taille incorrecte")
        return

    angle = 360 / cotes

    for i in range(cotes):

        avancer_cm(taille_cm)

        tourner(angle)
```

29. Utiliser `while`

Nous allons maintenant intégrer une boucle `while`.

Créer un compteur :

```
compteur = 0
```

Puis :

```
while compteur < 3:

    avancer_cm(20)

    compteur = compteur + 1
```

Le robot avance trois fois de :

```
20 cm
```

30. Comparer `while` et `for`

Ce programme :

```
compteur = 0
while compteur < 3:
    avancer_cm(20)
    compteur = compteur + 1
```

peut être remplacé par :

```
for i in range(3):
    avancer_cm(20)
```

Dans ce cas, `for` est généralement plus simple.

31. Pourquoi réviser `while` alors ?

Parce que bientôt nous utiliserons des conditions comme :

```
while distance > 200:
```

ou :

```
while not capteur.pressed():
```

Le nombre de répétitions ne sera alors pas connu à l'avance.

32. Défi 4 : escalier

Créer une fonction :

```
def escalier(nombre, taille_cm):
```

Le robot doit répéter :

```
Avancer
Tourner à droite
Avancer
Tourner à gauche
```

Solution possible

```
def escalier(nombre, taille_cm):

    for i in range(nombre):

        avancer_cm(taille_cm)

        tourner(90)

        avancer_cm(taille_cm)

        tourner(-90)
```

Tester :

```
escalier(3, 20)
```

33. Défi 5 : zigzag

Créer une fonction :

```
def zigzag(nombre, distance_cm):
```

Le robot doit alterner les rotations.

Une solution :

```
def zigzag(nombre, distance_cm):

    for i in range(nombre):

        avancer_cm(distance_cm)

        if i % 2 == 0:
            tourner(45)
        else:
            tourner(-45)
```

34. Nouvelle notion rapide : `%`

Dans :

```
i % 2
```

`%` donne le reste d'une division.

Par exemple :

```
4 divisé par 2

reste = 0
```

Donc :

```
4 % 2
```

donne :

```
0
```

Mais :

```
5 divisé par 2
```

```
reste = 1
```

Donc :

```
5 % 2
```

donne :

```
1
```

35. Détecter un nombre pair

Nous pouvons donc écrire :

```
if i % 2 == 0:
```

pour savoir si `i` est pair.

Cela permet d'alterner deux comportements.

36. Exemple du zigzag

Lorsque :

```
i = 0
```

le robot tourne de :

```
45°
```

Lorsque :

```
i = 1
```

il tourne de :

```
moins 45°
```

Puis :

```
i = 2
```

donne de nouveau :

```
45°
```

Le robot alterne donc automatiquement.

37. Défi 6 : spirale simple

Créer une trajectoire qui augmente progressivement la distance.

Par exemple :

```
10 cm

20 cm

30 cm

40 cm

50 cm
```

avec une rotation de :

```
90°
```

entre chaque déplacement.

Une solution :

```
distance = 10

for i in range(5):

    avancer_cm(distance)

    tourner(90)

    distance = distance + 10
```

38. Version avec la variable de boucle

Nous pouvons également écrire :

```
for i in range(1, 6):

    distance = i * 10

    avancer_cm(distance)

    tourner(90)
```

Les distances deviennent :

```
10

20

30

40

50
```

39. Défi 7 : parcours paramétrable

Créer une fonction :

```
def parcours(longueur_cm, largeur_cm, repetitions):
```

Elle doit répéter :

```
avancer longueur

tourner à droite

avancer largeur

tourner à gauche
```

Solution :

```
def parcours(longueur_cm, largeur_cm, repetitions):
    for i in range(repetitions):
        avancer_cm(longueur_cm)
        tourner(90)
        avancer_cm(largeur_cm)
        tourner(-90)
```

40. Calculer la distance totale

Créer :

```
def distance_parcours(longueur_cm, largeur_cm, repetitions):
```

Solution :

```
def distance_parcours(longueur_cm, largeur_cm, repetitions):
    distance = (longueur_cm + largeur_cm) * repetitions
    return distance
```

Puis :

```
total = distance_parcours(50, 30, 4)
print(total)
```

Résultat :

```
320 cm
```

41. Combiner calcul et mouvement

Nous pouvons écrire :

```
longueur = 50
largeur = 30
repetitions = 4

total = distance_parcours(
    longueur,
    largeur,
    repetitions
)

print("Distance totale :")
print(total)

parcours(
    longueur,
    largeur,
    repetitions
)
```

Le programme commence par calculer la distance prévue.

Puis le robot réalise le parcours.

42. Projet principal

Le terrain de l'exercice peut être organisé avec du ruban adhésif.

Par exemple :



Tu dois programmer le parcours sans déplacer manuellement le robot entre les différentes étapes.

43. Contraintes du projet

Le programme doit obligatoirement utiliser :

1. au moins 5 variables
2. au moins une fonction avec un paramètre
3. au moins une fonction avec deux paramètres
4. au moins une boucle `for`
5. au moins une condition `if`
6. au moins une utilisation de `return`
7. `DriveBase`
8. au moins 3 fonctions différentes créées par toi

44. Interdiction

Pour ce projet, il est interdit de programmer tout le parcours uniquement comme ceci :

```
robot.straight(600)
robot.turn(90)
robot.straight(400)
robot.turn(180)
robot.straight(400)
```

Même si cela fonctionne.

L'objectif est de montrer que tu sais organiser ton programme.

45. Exemple d'organisation attendue

```
# IMPORTS


# CONFIGURATION DU ROBOT


# VARIABLES


# FONCTIONS


# PROGRAMME PRINCIPAL
```

46. Exemple complet

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


def cm_vers_mm(cm):

    return cm * 10


def avancer_cm(cm):

    robot.straight(cm_vers_mm(cm))


def tourner(angle):

    robot.turn(angle)


def demi_tour():

    tourner(180)


def carre(taille_cm):

    for i in range(4):

        avancer_cm(taille_cm)

        tourner(90)


def verifier_distance(distance_cm):

    if distance_cm <= 0:

        print("Distance incorrecte")

        return False

    return True


distance1 = 60
distance2 = 40
distance3 = 30
taille_carre = 25
```

```
if verifier_distance(distance1):
    avancer_cm(distance1)
    tourner(90)
    avancer_cm(distance2)
    demi_tour()
    avancer_cm(distance2)
    tourner(-90)
    avancer_cm(distance3)
    carre(taille_carre)
```

47. Comprendre `True` et `False`

Notre fonction :

```
def verifier_distance(distance_cm):
    if distance_cm <= 0:
        return False
    return True
```

renvoie une valeur booléenne.

Nous pouvons donc écrire :

```
if verifier_distance(distance1):
```

Cela signifie :

```
SI la distance est valide
ALORS exécuter le parcours
```

48. Test progressif

Il est fortement déconseillé de programmer immédiatement tout le parcours puis de lancer le robot.

Il vaut mieux tester progressivement.

Étape 1

Tester :

```
avancer_cm(60)
```

Étape 2

Ajouter :

```
tourner(90)
```

Étape 3

Ajouter :

```
avancer_cm(40)
```

Et ainsi de suite.

49. Pourquoi tester progressivement ?

Si le robot ne fait pas ce que nous attendons après 15 instructions, nous ne savons pas forcément où se trouve l'erreur.

Mais si chaque étape est testée séparément, il est beaucoup plus facile de trouver le problème.

C'est le principe du débogage.

50. Introduire le débogage

Un bug est un problème dans un programme.

Déboguer signifie :

```
trouver le problème

comprendre sa cause

le corriger
```

Exemple :

```
def avancer_cm(cm):
    robot.straight(cm)
```

Problème :

Nous avons oublié de convertir les centimètres en millimètres.

Si nous écrivons :

```
avancer_cm(50)
```

le robot avance seulement de :

```
50 mm
```

soit :

```
5 cm
```

au lieu de :

```
50 cm
```

51. Méthode de débogage

Face à un problème, vérifier dans cet ordre :

```
1. Le programme se lance t-il ?

2. Y a t-il un message d'erreur ?

3. Les ports sont-ils corrects ?

4. Les variables contiennent elles les bonnes valeurs ?

5. Les unités sont-elles correctes ?

6. Les fonctions reçoivent elles les bons arguments ?

7. Les rotations ont elles le bon signe ?

8. Le problème est-il logiciel ou mécanique ?
```

52. Utiliser `print()` pour déboguer

`print()` est très utile.

Par exemple :

```
def avancer_cm(cm):

    distance_mm = cm * 10

    print("Distance en cm :")
    print(cm)

    print("Distance en mm :")
    print(distance_mm)

    robot.straight(distance_mm)
```

Si nous appelons :

```
avancer_cm(50)
```

nous pouvons vérifier que :

```
distance en cm = 50

distance en mm = 500
```

53. Déboguer un angle

Nous pouvons aussi écrire :

```
def polygone(cotes, taille):

    angle = 360 / cotes

    print("Angle :")
    print(angle)

    for i in range(cotes):

        robot.straight(taille)

        robot.turn(angle)
```

Cela permet de vérifier le calcul avant même d'observer le mouvement.

54. Défi de débogage 1

Trouver l'erreur :

```
def cm_vers_mm(cm):
    return cm / 10
```

Correction :

```
def cm_vers_mm(cm):
    return cm * 10
```

55. Défi de débogage 2

Trouver l'erreur :

```
def carre(taille):

    for i in range(3):

        robot.straight(taille)

        robot.turn(90)
```

Un carré possède quatre côtés.

Correction :

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

56. Défi de débogage 3

Trouver l'erreur :

```
def triangle(taille):

    for i in range(3):

        robot.straight(taille)

        robot.turn(90)
```

Pour un triangle équilatéral, la rotation doit être proche de :

```
120°
```

Correction :

```
robot.turn(120)
```

57. Défi de débogage 4

Trouver l'erreur :

```
def tourner(angle):
robot.turn(angle)
```

Problème :

L'indentation est incorrecte.

Correction :

```
def tourner(angle):
    robot.turn(angle)
```

58. Défi de débogage 5

Trouver l'erreur :

```
def avancer(distance):
    robot.straight(distance)

avancer()
```

La fonction attend un argument.

Correction possible :

```
avancer(500)
```

59. Compétition facultative

Chaque étudiant possède le même terrain.

Le score peut être basé sur :

Critère	Points
Parcours terminé	10
Position finale	5
Orientation finale	5
Qualité du code	5
Utilisation des fonctions	5
Utilisation correcte des boucles	5
Lisibilité	5

Total :

```
40 points
```

60. Critères de qualité du code

Un bon programme doit notamment avoir :

```
des noms compréhensibles

des fonctions courtes

des valeurs stockées dans des variables

peu de répétitions inutiles

une indentation correcte

une organisation logique
```

61. Exemple mauvais code

```
robot.straight(500)
robot.turn(90)
robot.straight(300)
robot.turn(-90)
robot.straight(500)
robot.turn(90)
robot.straight(300)
robot.turn(-90)
robot.straight(500)
robot.turn(90)
robot.straight(300)
robot.turn(-90)
```

Le programme fonctionne peut être, mais il contient énormément de répétitions.

62. Exemple amélioré

```
def motif():

    robot.straight(500)

    robot.turn(90)

    robot.straight(300)

    robot.turn(-90)
```

Puis :

```
for i in range(3):
    motif()
```

Le programme est beaucoup plus simple à modifier.

63. Questions de fin de cours

Question 1

Pourquoi utilisons nous des fonctions ?

Question 2

Quelle différence existe entre `for` et `while` ?

Question 3

À quoi sert `return` ?

Question 4

Pourquoi `DriveBase` a t il besoin du diamètre des roues ?

Question 5

Pourquoi faut il mesurer le comportement réel du robot ?

Question 6

Qu'est ce qu'une erreur cumulative ?

Question 7

À quoi peut servir `print()` lors du développement d'un robot ?

Question 8

Que signifie déboguer ?

Correction

Réponse 1

Les fonctions permettent d'organiser et de réutiliser le code.

Réponse 2

`for` est particulièrement adapté lorsque le nombre de répétitions est connu.

`while` est adapté lorsque la répétition dépend d'une condition.

Réponse 3

`return` permet à une fonction de renvoyer une valeur.

Réponse 4

Pour convertir la rotation des moteurs en distance parcourue.

Réponse 5

Parce qu'un robot réel possède des imperfections mécaniques et ne suit pas toujours exactement les valeurs théoriques.

Réponse 6

Il s'agit de petites erreurs qui s'additionnent au fur et à mesure du déplacement.

Réponse 7

À afficher des variables et des résultats intermédiaires afin de comprendre ce que fait le programme.

Réponse 8

Trouver, comprendre et corriger une erreur dans le programme ou le fonctionnement du robot.

Cours 13 : Le capteur tactile avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le rôle d'un capteur tactile
2. connecter correctement le capteur à la brique EV3
3. initialiser un `TouchSensor`
4. utiliser `pressed()`
5. lire une information provenant du capteur
6. utiliser `if` avec un capteur
7. utiliser `while` avec un capteur
8. arrêter un robot lors d'une collision
9. créer un comportement d'évitement simple
10. comprendre le principe général d'un robot autonome

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 2 roues identiques
4. 1 capteur tactile EV3
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. plusieurs objets pouvant servir d'obstacles
8. un espace libre au sol

1. Introduction

Jusqu'à présent, notre robot suivait des instructions définies à l'avance.

Par exemple :

```
robot.straight(500)

robot.turn(90)

robot.straight(300)
```

Le robot exécute ces instructions sans savoir ce qui se trouve devant lui.

Si nous plaçons une boîte sur son chemin, il risque simplement de rentrer dedans.

Pour créer un robot autonome, nous devons lui permettre d'observer son environnement.

C'est le rôle des capteurs.

2. Le principe d’un robot autonome

On peut représenter le fonctionnement d’un robot ainsi :

```
CAPTEUR
↓
INFORMATION
↓
PROGRAMME
↓
DÉCISION
↓
ACTION
```

Par exemple :

```
Le capteur tactile est appuyé
↓
Le programme détecte une collision
↓
Le robot s’arrête
```

3. Qu’est-ce qu’un capteur ?

Un capteur est un composant qui permet au robot d’obtenir une information.

Nous utiliserons plusieurs types de capteurs au cours des prochaines leçons.

```
Capteur tactile

Capteur ultrason

Capteur couleur

Capteur de lumière

Gyroscope
```

Chaque capteur mesure quelque chose de différent.

4. Le capteur tactile

Le capteur tactile fonctionne comme un bouton.

Il possède principalement deux états.

```
Appuyé

ou

Non appuyé
```



Nous pouvons donc représenter son état avec :

```
True
```

```
ou
```

```
False
```

5. Connexion du capteur

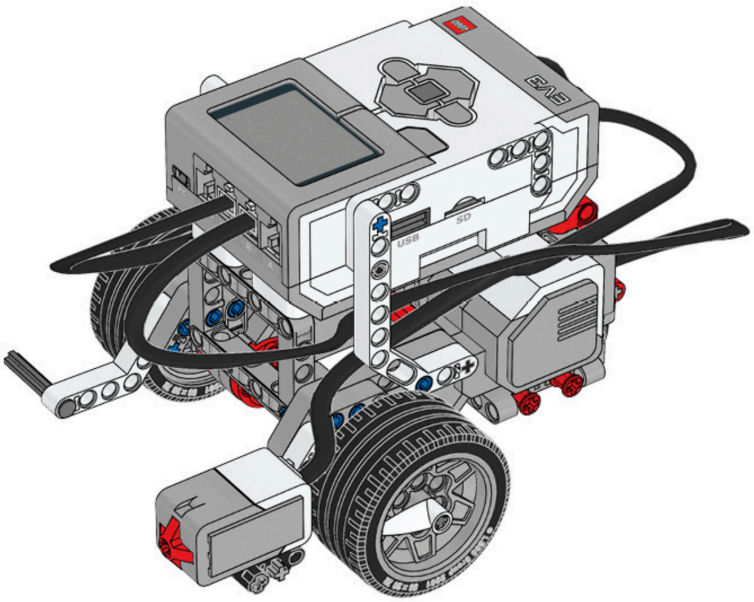
Les capteurs EV3 sont généralement connectés aux ports :

```
1
2
3
4
```

Pour ce cours, nous utiliserons :

```
Port 1 = capteur tactile
```

Nous utiliserons le robot construit au **cours 8** auquel on ajoute le capteur tactile.



6. Importer `TouchSensor`

Nous devons importer :

```
from pybricks.ev3devices import TouchSensor
```

Nous avons également besoin de :

```
from pybricks.parameters import Port
```

Puis :

```
capteur = TouchSensor(Port.S1)
```

7. Premier programme

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port
capteur = TouchSensor(Port.S1)
print(capteur.pressed())
```

Lancer une première fois le programme sans toucher au capteur.

Puis lancer le programme en maintenant le bouton appuyé.

Observer la différence.

8. La fonction `pressed()`

La commande :

```
capteur.pressed()
```

renvoie :

```
True
```

si le bouton est appuyé.

Et :

```
False
```

si le bouton n'est pas appuyé.

Nous pouvons donc stocker cette information.

```
etat = capteur.pressed()
print(etat)
```

9. Utiliser `if`

Nous pouvons maintenant écrire :

```
if capteur.pressed():
    print("Collision")
```

Si le bouton n'est pas appuyé, rien ne se passe.

10. Ajouter `else`

```
if capteur.pressed():
    print("Collision")
else:
    print("Aucune collision")
```

Le programme prend maintenant une décision.

11. Exercice pratique 1

Créer un programme qui affiche :

```
STOP
```

lorsque le bouton est appuyé.

Sinon, il doit afficher :

```
AVANCER
```

Solution :

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port

capteur = TouchSensor(Port.S1)

if capteur.pressed():
    print("STOP")
else:
    print("AVANCER")
```

12. Le problème

Le programme précédent ne lit le capteur qu'une seule fois.

Si le bouton est libre au démarrage, le programme affiche :

```
AVANCER
```

puis se termine.

Nous voulons plutôt surveiller le capteur continuellement.

13. Utiliser `while True`

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():
        print("Collision")
    else:
        print("Aucune collision")

    wait(50)
```

Le capteur est maintenant vérifié continuellement.

14. Pourquoi ajouter une petite attente ?

Sans :

```
wait(50)
```

la boucle s'exécute aussi vite que possible.

Pour une simple surveillance, une petite pause peut être suffisante.

Ici :

```
50 ms
```

correspond à :

```
0,05 seconde
```

Le robot vérifie donc l'état très régulièrement.

15. Ajouter la brique EV3

Nous pouvons maintenant faire produire un son lorsqu'une collision est détectée.

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

ev3 = EV3Brick()

capteur = TouchSensor(Port.S1)

while True:

    if capteur.pressed():
        ev3.speaker.beep()

    wait(50)
```

16. Un problème apparaît

Si nous maintenons le capteur appuyé pendant une seconde, le robot peut produire plusieurs sons.

Pourquoi ?

Parce que la boucle vérifie continuellement :

```
Capteur appuyé ?

Oui

BIP

Recommencer

Capteur appuyé ?

Oui

BIP
```

La même action est répétée tant que le bouton reste appuyé.

17. Attendre le relâchement

Une solution simple consiste à attendre que le bouton soit relâché.

```
while True:

    if capteur.pressed():

        ev3.speaker.beep()

        while capteur.pressed():
            wait(10)
```

Après le son, le programme attend que le capteur soit libéré avant de continuer.

18. Exercice pratique 2

Créer un compteur de pressions.

Chaque fois que le capteur est appuyé puis relâché, le compteur augmente de `1`.

Exemple attendu :

```
1
2
3
4
5
```

Une solution possible :

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur = TouchSensor(Port.S1)

compteur = 0

while True:

    if capteur.pressed():

        compteur = compteur + 1

        print(compteur)

        while capteur.pressed():
            wait(10)

    wait(10)
```

19. Application au robot mobile

Nous allons maintenant connecter notre capteur tactile à l'avant du robot.

Configuration :

```
Moteur gauche = B

Moteur droit = C

Capteur tactile = 1
```

20. Initialiser le robot

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur = TouchSensor(Port.S1)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

21. Avancer jusqu'à une collision

Nous pouvons écrire :

```
robot.drive(200, 0)

while not capteur.pressed():
    pass

robot.stop()
```

Le robot avance jusqu'à ce que le capteur soit appuyé.

Puis il s'arrête.

22. Comprendre la condition

```
while not capteur.pressed():
```

signifie :

```
TANT QUE

le capteur n'est pas appuyé
```

Le robot continue donc à avancer.

Lorsque le capteur est appuyé :

```
not True
```

devient :

```
False
```

La boucle se termine.

23. Exercice pratique 3

Programmer :

```
Avancer

↓

Toucher un obstacle

↓

STOP
```

Solution :

```
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur = TouchSensor(Port.S1)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

robot.drive(200, 0)

while not capteur.pressed():
    pass

robot.stop()
```

24. Ajouter une réaction

Arrêter n'est pas très intéressant.

Nous pouvons demander au robot de :

```
avancer

↓

collision

↓

reculer

↓

arrêt
robot.drive(200, 0)

while not capteur.pressed():
    pass

robot.stop()

robot.straight(-200)
```

Le robot recule de :

```
200 mm
```

après la collision.

25. Ajouter une rotation

Nous pouvons maintenant créer :

```
AVANCER

↓

COLLISION

↓

RECULER

↓

TOURNER

↓

STOP
robot.drive(200, 0)

while not capteur.pressed():
    pass

robot.stop()

robot.straight(-200)

robot.turn(90)
```

26. Premier comportement autonome

Nous pouvons mettre tout cela dans une boucle infinie.

```
while True:

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()

    robot.straight(-200)

    robot.turn(90)
```

Le comportement devient :

```
Avancer
↓
Chercher un obstacle
↓
Collision
↓
Reculer
↓
Tourner
↓
Avancer à nouveau
```

27. Pourquoi ce robot est-il autonome ?

Parce que le moment de la réaction n'est pas défini à l'avance.

Nous ne lui demandons pas :

```
Avance pendant 3 secondes
```

Nous lui demandons :

```
Avance jusqu'à ce qu'un événement se produise
```

Le robot adapte donc son comportement à son environnement.

28. Ajouter un son

```
from pybricks.hubs import EV3Brick

ev3 = EV3Brick()
```

Puis :

```
if capteur.pressed():
    ev3.speaker.beep()
```

Dans notre comportement :

```
while True:

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()

    ev3.speaker.beep()

    robot.straight(-200)

    robot.turn(90)
```

29. Ajouter des fonctions

Nous pouvons rendre le programme plus lisible.

```
def avancer_jusqua_collision():

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()
```

Puis :

```
def eviter_obstacle():

    robot.straight(-200)

    robot.turn(90)
```

Programme principal :

```
while True:

    avancer_jusqua_collision()

    eviter_obstacle()
```

30. Comprendre l'abstraction

Le programme principal devient :

```
avancer jusqu'à collision

éviter obstacle

recommencer
```

Alors que chaque fonction peut contenir plusieurs instructions.

C'est exactement l'intérêt des fonctions vues précédemment.

31. Exercice pratique 4

Créer trois fonctions :

```
def avancer():
def reculer():
def collision():
```

Une solution possible :

```
def avancer():

    robot.drive(200, 0)


def reculer():

    robot.straight(-200)


def collision():

    return capteur.pressed()
```

Puis :

```
avancer()

while not collision():
    pass

robot.stop()

reculer()
```

32. Fonction retournant une valeur

Cette fonction :

```
def collision():
    return capteur.pressed()
```

retourne :

```
True
```

ou :

```
False
```

Nous pouvons donc écrire :

```
if collision():
```

ou :

```
while not collision():
```

33. Créer un compteur de collisions

Nous pouvons compter combien d'obstacles le robot a rencontrés.

```
compteur = 0

while True:

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()

    compteur = compteur + 1

    print(compteur)

    robot.straight(-200)

    robot.turn(90)
```

34. Défi pratique 1

Le robot doit s'arrêter définitivement après :

```
3 collisions
```

Structure possible :

```
compteur = 0

while compteur < 3:

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()

    compteur = compteur + 1

    robot.straight(-200)

    robot.turn(90)

robot.stop()
```

35. Ajouter un son différent à la fin

```
for i in range(3):
    ev3.speaker.beep()
```

Nous pouvons donc annoncer la fin du programme après trois collisions.

36. Défi pratique 2

Créer ce comportement :

```
Avancer
↓
Collision 1
↓
Reculer
↓
Tourner à droite
↓
Avancer
↓
Collision 2
↓
Reculer
↓
Tourner à gauche
↓
Avancer
```

Il faut alterner le sens des rotations.

37. Solution avec %

Nous pouvons utiliser le compteur.

```
compteur = 0

while True:

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()

    compteur = compteur + 1

    robot.straight(-200)

    if compteur % 2 == 0:
        robot.turn(-90)
    else:
        robot.turn(90)
```

Le robot alterne automatiquement.

38. Visualiser l'alternance

```
Collision 1

1 % 2 = 1
↓
droite
Collision 2

2 % 2 = 0
↓
gauche
Collision 3
↓
droite
```

Et ainsi de suite.

39. Gérer le rebond du bouton

Un bouton mécanique peut parfois changer rapidement d'état lors d'un appui.

Pour des exercices simples, ce problème sera limité.

Mais après une collision, il est utile de s'assurer que le robot s'est suffisamment éloigné de l'obstacle avant de recommencer.

C'est aussi pour cela que nous effectuons :

```
robot.straight(-200)
```

avant de reprendre le déplacement.

40. Tester différentes distances de recul

Tester :

```
50 mm
```

```
100 mm
```

```
200 mm
```

```
300 mm
```

Noter le comportement.

Distance de recul	Résultat
50 mm	
100 mm	
200 mm	
300 mm	

Question :

Quelle distance permet au robot de se dégager correctement sans perdre trop de temps ?

41. Tester différentes rotations

Tester après collision :

```
robot.turn(45)
```

puis :

```
robot.turn(90)
```

puis :

```
robot.turn(135)
```

Observer le comportement.

42. Pourquoi 90 degrés n’est pas toujours idéal ?

Un angle de 90 degrés peut fonctionner dans un environnement simple.

Mais dans une pièce avec plusieurs obstacles, il peut parfois placer immédiatement le robot face à un autre objet.

Nous pourrons plus tard utiliser des stratégies plus avancées.

43. Introduire l’aléatoire

Une stratégie possible consiste à choisir différents angles.

Nous pouvons par exemple définir plusieurs valeurs possibles manuellement.

Mais pour ce cours, l’objectif principal reste de comprendre le fonctionnement du capteur.

L’aléatoire pourra être abordé plus tard comme extension.

44. Mini défi 3 : le robot prisonnier

Créer une zone fermée avec des boîtes.

Le robot est placé à l’intérieur.

Son comportement :

```
Avancer
↓
Collision
↓
Reculer
↓
Tourner
↓
Repartir
```

Le robot doit continuer à se déplacer sans intervention humaine pendant au moins :

```
1 minute
```

45. Programme possible

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor, TouchSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

ev3 = EV3Brick()

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur = TouchSensor(Port.S1)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

compteur = 0

while True:

    robot.drive(200, 0)

    while not capteur.pressed():
        pass

    robot.stop()

    ev3.speaker.beep()

    compteur = compteur + 1

    print(compteur)

    robot.straight(-200)

    if compteur % 2 == 0:
        robot.turn(-90)
    else:
        robot.turn(90)
```

46. Limite du capteur tactile

Le capteur tactile possède une limitation évidente.

Le robot doit :

```
toucher l'obstacle
```

pour savoir qu'il existe.

Il ne peut pas détecter un mur situé à :
`50 cm`
devant lui.

Pour cela, nous aurons besoin d'un autre capteur.

47. Différence entre détection et anticipation

Avec le capteur tactile :

```
Obstacle

↓

CONTACT

↓

Détection
```

Le robot réagit après contact.

Avec un capteur de distance :

```
Obstacle

↓

Mesure de distance

↓

Détection avant le contact
```

Le robot pourra donc anticiper.

48. Questions de fin de cours

Question 1

Que mesure le capteur tactile ?

Question 2

Que retourne :

```
capteur.pressed()
```

Question 3

Que signifie :

```
while not capteur.pressed():
```

Question 4

Pourquoi avons nous besoin d'une boucle pour surveiller le capteur ?

Question 5

Quelle différence existe entre :

```
if capteur.pressed():
```

et :

```
while capteur.pressed():
```

Question 6

Pourquoi le robot doit il souvent reculer après une collision ?

Question 7

Pourquoi peut on considérer notre robot anticollision comme autonome ?

Question 8

Quelle est la principale limitation du capteur tactile ?

Correction

Réponse 1

Il détecte si son bouton est appuyé ou non.

Réponse 2

`True`

ou :

`False`

Réponse 3

Répéter les instructions tant que le capteur n'est pas appuyé.

Réponse 4

Parce que l'environnement peut changer après le lancement du programme.

Il faut donc lire régulièrement le capteur.

Réponse 5

`if` effectue un test.

`while` répète tant que la condition reste vraie.

Réponse 6

Pour se dégager de l'obstacle et libérer le capteur.

Réponse 7

Parce que son comportement dépend d'informations provenant de son environnement et non uniquement d'une séquence prédéfinie.

Réponse 8

Il doit entrer physiquement en contact avec l'obstacle.

Difficultés fréquentes

Erreur 1 : mauvais port

Le programme indique :

```
TouchSensor(Port.S1)
```

mais le capteur est connecté au port 2.

Le programme ne fonctionnera pas correctement.

Erreur 2 : oublier les parenthèses

Incorrect :

```
if capteur.pressed:
```

Correct :

```
if capteur.pressed():
```

`pressed` est une méthode.

Il faut l'appeler avec les parenthèses.

Erreur 3 : oublier `not`

Tu veux avancer jusqu'à la collision et si tu écris :

```
while capteur.pressed():
```

Mais au départ, le capteur n'est pas appuyé.

La boucle ne démarre donc pas.

Pour attendre la collision :

```
while not capteur.pressed():
```

Erreur 4 : oublier d'arrêter `drive()`

Programme problématique :

```
robot.drive(200, 0)

while not capteur.pressed():
    pass

robot.straight(-200)
```

Il est préférable d'arrêter le mouvement continu avant de lancer une autre action :

```
robot.stop()
```

Cours 14 : Le capteur ultrason avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le rôle d'un capteur ultrason
2. mesurer une distance
3. utiliser `UltrasonicSensor`
4. utiliser `distance()`
5. prendre une décision selon une distance
6. faire varier la vitesse du robot
7. arrêter le robot avant une collision
8. créer un robot qui maintient une distance
9. filtrer des mesures simples
10. combiner capteur ultrason, conditions et boucles

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur ultrason EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. plusieurs obstacles
8. une règle ou un mètre
9. un espace libre au sol

1. Introduction

Lors du cours précédent, nous avons utilisé le capteur tactile.

Le robot pouvait détecter un obstacle uniquement après l'avoir touché.

Le comportement était donc :

```
AVANCER
↓
CONTACT
↓
DÉTECTION
↓
STOP
```

Aujourd'hui, nous allons améliorer notre robot.

Nous voulons qu'il puisse détecter un obstacle avant de le toucher.

Le comportement devient :

```
AVANCER
↓
MESURER LA DISTANCE
↓
OBSTACLE PROCHE
↓
STOP
```

Pour cela, nous allons utiliser le capteur ultrason.

2. Qu’est-ce qu’un capteur ultrason ?

Le capteur ultrason permet de mesurer approximativement la distance entre le robot et un objet.

Il fonctionne en envoyant une onde sonore à très haute fréquence.

Cette onde se déplace jusqu’à un obstacle.

Elle rebondit ensuite vers le capteur.

Le principe peut être représenté ainsi :

```
CAPTEUR

→ → → onde sonore → → →

OBSTACLE

← ← ← écho ← ← ←

CAPTEUR
```



Le capteur mesure le temps nécessaire pour recevoir l’écho.

À partir de ce temps, il estime la distance.

3. Une comparaison avec l’écho

Si nous crions dans une grande vallée, nous pouvons parfois entendre notre voix revenir quelques instants plus tard.

C’est un écho.

Le capteur ultrason utilise un principe similaire, mais avec des sons que nous n’entendons normalement pas.

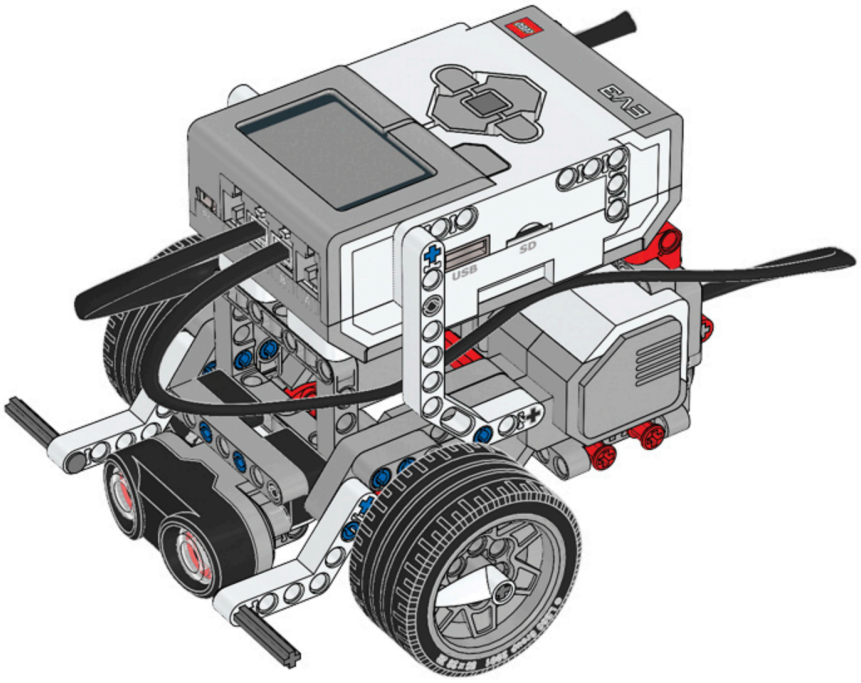
4. Connexion du capteur

Pour ce cours, nous utiliserons :

```
Port 4 = capteur ultrason
```

Notre configuration sera donc :

```
Port B = moteur gauche
Port C = moteur droit
Port 4 = capteur ultrason
```



5. Importer le capteur

Nous devons importer :

```
from pybricks.ev3devices import UltrasonicSensor
```

Puis :

```
from pybricks.parameters import Port
```

Et initialiser le capteur :

```
capteur_ultrason = UltrasonicSensor(Port.S4)
```

6. Premier programme

```
from pybricks.ev3devices import UltrasonicSensor
from pybricks.parameters import Port

capteur_ultrason = UltrasonicSensor(Port.S4)
distance = capteur_ultrason.distance()
print(distance)
```

Le programme affiche la distance mesurée.

7. Quelle unité est utilisée ?

La fonction :

```
capteur_ultrason.distance()
```

renvoie une distance en :

```
millimètres
```

Donc :

```
100 mm = 10 cm

200 mm = 20 cm

500 mm = 50 cm

1000 mm = 1 mètre
```

8. Première expérience

Placer un objet devant le capteur à environ :

```
10 cm
```

puis :

```
20 cm
```

puis :

```
50 cm
```

puis :

```
100 cm
```

Afficher la mesure obtenue.

Distance réelle	Distance mesurée
100 mm	
200 mm	
500 mm	
1000 mm	

9. Observer continuellement la distance

Pour surveiller la distance en permanence :

```
from pybricks.ev3devices import UltrasonicSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur_ultrason = UltrasonicSensor(Port.S4)

while True:

    distance = capteur_ultrason.distance()

    print(distance)

    wait(100)
```

Déplacer lentement un objet devant le capteur.

Observer les valeurs.

10. Utiliser `if`

Nous pouvons maintenant créer une condition.

```
distance = capteur_ultrason.distance()

if distance < 200:
    print("Obstacle proche")
```

Cela signifie :

```
SI distance inférieure à 200 mm

ALORS afficher obstacle proche
```

Donc :

```
200 mm = 20 cm
```

11. Ajouter `else`

```
distance = capteur_ultrason.distance()

if distance < 200:
    print("Obstacle proche")
else:
    print("Zone libre")
```

12. Exercice pratique 1

Créer un programme qui affiche :

```
DANGER
```

si un obstacle est à moins de :

```
15 cm
```

Sinon afficher :

```
OK
```

Il faut d'abord convertir :

```
15 cm = 150 mm
```

Solution :

```
distance = capteur_ultrason.distance()

if distance < 150:
    print("DANGER")
else:
    print("OK")
```

13. Utiliser plusieurs zones

Nous pouvons utiliser `elif`.

```
distance = capteur_ultrason.distance()

if distance < 150:
    print("STOP")

elif distance < 400:
    print("RALENTIR")

else:
    print("AVANCER")
```

Nous avons maintenant trois comportements.

14. Comprendre les zones

```
0 à 149 mm
↓
STOP
150 à 399 mm
↓
RALENTIR
400 mm ou plus
↓
AVANCER
```

15. Pourquoi l’ordre est important ?

Considérons :

```
if distance < 400:
    print("RALENTIR")

elif distance < 150:
    print("STOP")
```

Si la distance vaut :

```
100 mm
```

alors :

```
100 < 400
```

est déjà vrai.

Le programme affiche donc :

```
RALENTIR
```

et ne vérifie jamais la deuxième condition.

Il faut commencer par la condition la plus restrictive.

```
if distance < 150:
    print("STOP")

elif distance < 400:
    print("RALENTIR")
```

16. Ajouter le robot

Créons notre `DriveBase`.

```
from pybricks.ev3devices import Motor, UltrasonicSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_ultrason = UltrasonicSensor(Port.S4)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

17. Avancer jusqu'à un obstacle

Nous voulons :

```
AVANCER
↓
MESURER
↓
DISTANCE < 200 mm
↓
STOP
```

Programme :

```
robot.drive(200, 0)

while capteur_ultrason.distance() >= 200:
    pass

robot.stop()
```

Le robot avance tant que l'obstacle reste à au moins 200 mm.

18. Comparaison avec le capteur tactile

Capteur tactile :

```
while not capteur.pressed():
```

Capteur ultrason :

```
while capteur_ultrason.distance() >= 200:
```

Dans les deux cas, nous attendons un événement.

Mais l'événement est différent.

Avec le tactile :

```
contact
```

Avec l'ultrason :

```
distance trop faible
```

19. Exercice pratique 2

Faire avancer le robot jusqu'à ce qu'il se trouve à environ :

```
30 cm
```

d'un obstacle.

Solution :

```
robot.drive(200, 0)

while capteur_ultrason.distance() >= 300:
    pass

robot.stop()
```

20. Tester plusieurs distances

Tester :

```
100 mm

200 mm

300 mm

500 mm
```

Observer l'endroit où le robot s'arrête.

Distance demandée	Distance réelle après arrêt
100 mm	
200 mm	
300 mm	
500 mm	

21. Pourquoi la distance réelle n'est-elle pas parfaite ?

Lorsque le programme détecte la limite, le robot est déjà en mouvement.

Il possède donc une certaine inertie.

Il peut continuer à avancer légèrement avant l'arrêt complet.

La vitesse influence donc la précision.

22. Expérience sur la vitesse

Tester le même arrêt à :

```
robot.drive(100, 0)
```

puis :

```
robot.drive(300, 0)
```

puis :

```
robot.drive(500, 0)
```

Observer la précision de l'arrêt.

Question :

Le robot rapide s'arrête il à la même distance que le robot lent ?

23. Faire ralentir progressivement

Nous pouvons créer plusieurs zones.

```
while True:
    distance = capteur_ultrason.distance()
    if distance < 150:
        robot.stop()
    elif distance < 400:
        robot.drive(100, 0)
    else:
        robot.drive(300, 0)
```

Le robot ralentit à l'approche de l'obstacle.

24. Comprendre le comportement

Si l'obstacle est loin :

```
distance = 800 mm
↓
300 mm/s
```

Lorsque le robot approche :

```
distance = 300 mm
↓
100 mm/s
```

Puis :

```
distance = 100 mm
↓
STOP
```

C'est déjà un comportement beaucoup plus intelligent.

25. Exercice pratique 3

Créer quatre zones.

```
Distance supérieure à 600 mm
↓
vitesse = 400 mm/s
Distance entre 300 et 600 mm
↓
vitesse = 250 mm/s
Distance entre 150 et 300 mm
↓
vitesse = 100 mm/s
Distance inférieure à 150 mm
↓
STOP
```

Une solution possible :

```
while True:

    distance = capteur_ultrason.distance()

    if distance < 150:

        robot.stop()

    elif distance < 300:

        robot.drive(100, 0)

    elif distance < 600:

        robot.drive(250, 0)

    else:

        robot.drive(400, 0)
```

26. Afficher la distance

Pendant les tests, il peut être utile d'ajouter :

```
print(distance)
```

Programme :

```
while True:

    distance = capteur_ultrason.distance()

    print(distance)

    if distance < 150:

        robot.stop()

    elif distance < 300:

        robot.drive(100, 0)

    else:

        robot.drive(300, 0)
```

Cela facilite le débogage.

27. Créer une fonction

```
def mesurer_distance():
    return capteur_ultrason.distance()
```

Nous pouvons ensuite écrire :

```
distance = mesurer_distance()
```

Cela rend le programme plus lisible.

28. Fonction pour déterminer une vitesse

```
def choisir_vitesse(distance):

    if distance < 150:
        return 0

    elif distance < 300:
        return 100

    elif distance < 600:
        return 250

    else:
        return 400
```

Programme principal :

```
while True:

    distance = mesurer_distance()

    vitesse = choisir_vitesse(distance)

    robot.drive(vitesse, 0)
```

29. Pourquoi cette version est intéressante ?

Nous avons séparé :

```
MESURER
```

de :

```
DÉCIDER
```

et de :

```
AGIR
```

Cela correspond exactement à :

```
PERCEVOIR

↓

DÉCIDER

↓

AGIR
```

30. Arrêter proprement avec une condition

Nous pouvons aussi sortir d'une boucle.

```
while True:

    distance = capteur_ultrason.distance()

    if distance < 200:
        break

    robot.drive(200, 0)

robot.stop()
```

31. Comparaison avec une condition dans `while`

Cette version :

```
while capteur_ultrason.distance() >= 200:
    robot.drive(200, 0)

robot.stop()
```

est plus compacte.

La version avec `break` est souvent intéressante lorsque plusieurs conditions différentes peuvent arrêter le robot.

32. Éviter un obstacle

Nous pouvons maintenant reprendre le comportement du capteur tactile, mais sans collision.

```
while True:

    robot.drive(200, 0)

    while capteur_ultrason.distance() > 200:
        pass

    robot.stop()

    robot.straight(-100)

    robot.turn(90)
```

Le robot détecte l'obstacle avant de le toucher.

33. Défi pratique 4

Créer le comportement :

```
Avancer
↓
Obstacle à moins de 25 cm
↓
STOP
↓
Reculer de 10 cm
↓
Tourner de 90°
↓
Repartir
```

Une solution possible :

```
while True:

    robot.drive(200, 0)

    while capteur_ultrason.distance() >= 250:
        pass

    robot.stop()

    robot.straight(-100)

    robot.turn(90)
```

34. Compter les obstacles

```
compteur = 0

while compteur < 5:

    robot.drive(200, 0)

    while capteur_ultrason.distance() >= 250:
        pass

    robot.stop()

    compteur = compteur + 1

    print(compteur)

    robot.straight(-100)

    robot.turn(90)
```

Après cinq obstacles, le robot s'arrête.

35. Défi pratique 5

Faire produire un son à chaque obstacle.
Après le cinquième obstacle, produire trois sons.

Structure :

```
compteur = 0

while compteur < 5:

    # déplacement

    # détection

    ev3.speaker.beep()

    compteur = compteur + 1

    # évitement

for i in range(3):
    ev3.speaker.beep()
```

36. Suivre un objet

Le capteur ultrason permet aussi de créer un robot qui essaie de rester à une certaine distance d'un objet.

Par exemple :

```
distance cible = 300 mm
```

Si l'objet est trop loin :

```
AVANCER
```

S'il est trop proche :

```
RECULER
```

S'il est à la bonne distance :

```
STOP
```

37. Premier robot suiveur

```
distance_cible = 300
while True:
    distance = capteur_ultrason.distance()
    if distance > 350:
        robot.drive(150, 0)
    elif distance < 250:
        robot.drive(-150, 0)
    else:
        robot.stop()
```

38. Comprendre la zone morte

Nous n'avons pas utilisé uniquement :

```
300 mm
```

Nous avons créé une zone :

```
250 mm à 350 mm
```

Dans cette zone, le robot s'arrête.

Pourquoi ?

Parce que si nous demandions exactement :

```
300 mm
```

le robot pourrait continuellement faire :

```
avancer

reculer

avancer

reculer
```

à cause des petites variations de mesure.

39. Cette zone s’appelle une tolérance

Nous pouvons définir :

```
distance_cible = 300
tolerance = 50
```

Puis :

```
if distance > distance_cible + tolerance:
```

et :

```
elif distance < distance_cible - tolerance:
```

40. Version plus propre

```
distance_cible = 300
tolerance = 50
vitesse = 150

while True:

    distance = capteur_ultrason.distance()

    if distance > distance_cible + tolerance:

        robot.drive(vitesse, 0)

    elif distance < distance_cible - tolerance:

        robot.drive(-vitesse, 0)

    else:

        robot.stop()
```

41. Exercice pratique 6

Modifier les valeurs pour maintenir environ :

```
50 cm
```

de distance.

Avec une tolérance de :

```
5 cm
```

Solution :

```
distance_cible = 500
tolerance = 50
```

42. Comprendre l'erreur

Nous pouvons aller plus loin.

Si nous voulons :

```
distance cible = 300 mm
```

et que nous mesurons :

```
distance = 500 mm
```

nous pouvons calculer :

```
erreur = distance mesurée moins distance cible
```

Donc :

```
500 moins 300 = 200 mm
```

43. Calcul en Python

```
distance_cible = 300
distance = capteur_ultrason.distance()
erreur = distance - distance_cible
print(erreur)
```

Si l'erreur est positive :
```
objet trop loin
```

Si l'erreur est négative :
```
objet trop proche
```

Si l'erreur est proche de zéro :
```
bonne distance
```

44. Pourquoi cette notion est importante ?

Parce que plus tard, nous utiliserons exactement le même principe pour le suivi de ligne.

Nous aurons :

```
valeur souhaitée

moins

valeur mesurée

=

erreur
```

Puis nous utiliserons cette erreur pour corriger le robot.

C'est la base de la régulation.

45. Créer une vitesse proportionnelle à l'erreur

Voici une première idée :

```
distance_cible = 300

while True:

    distance = capteur_ultrason.distance()

    erreur = distance - distance_cible

    vitesse = erreur
```

Si :

```
erreur = 200
```

alors :

```
vitesse = 200
```

Si :

```
erreur = 50
```

alors :

```
vitesse = 50
```

Le robot ralentit donc lorsqu'il approche de la distance souhaitée.

46. Programme proportionnel simple

```
distance_cible = 300

while True:

    distance = capteur_ultrason.distance()

    erreur = distance - distance_cible

    vitesse = erreur

    robot.drive(vitesse, 0)
```

Attention : ce programme est surtout destiné à illustrer le principe.

Dans la pratique, il faudra limiter la vitesse.

47. Limiter la vitesse

Nous pouvons créer :

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

Puis :

```
vitesse = limiter(erreur, -300, 300)
```

48. Programme amélioré

```
distance_cible = 300

def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur


while True:

    distance = capteur_ultrason.distance()

    erreur = distance - distance_cible

    vitesse = limiter(erreur, -300, 300)

    robot.drive(vitesse, 0)
```

49. Introduire un coefficient

Nous pouvons écrire :

```
vitesse = erreur * 0.5
```

Le nombre :

```
0,5
```

détermine la force de la correction.

Nous pouvons appeler cette valeur :

```
kp
```

Par exemple :

```
kp = 0.5
```

```
vitesse = erreur * kp
```

Nous retrouverons cette notion dans les prochains cours sur le suivi de ligne.

50. Expérience sur `kp`

Tester :

```
kp = 0.2
```

puis :

```
kp = 0.5
```

puis :

```
kp = 1
```

Observer comment le robot réagit.

Un `kp` faible donne généralement une réaction plus douce.

Un `kp` élevé provoque une réaction plus forte.

51. Limites du capteur ultrason

Le capteur ultrason n’est pas parfait.

Certaines situations peuvent donner des mesures moins fiables.

Par exemple :

```
objet très petit

objet incliné

surface absorbant le son

plusieurs robots ultrason proches

angles particuliers

obstacle hors de l’axe du capteur
```

Il faut donc toujours tester dans les conditions réelles.

52. Mesures instables

Parfois, plusieurs lectures successives peuvent être légèrement différentes.

Par exemple :

```
300

304

299

307

302
```

Même si l’objet n’a presque pas bougé.

C’est normal pour un capteur réel.

53. Moyenne simple

Nous pouvons mesurer plusieurs fois.

```
somme = 0

for i in range(5):

    somme = somme + capteur_ultrason.distance()

moyenne = somme / 5

print(moyenne)
```

Une moyenne peut réduire l’influence de petites variations.

54. Créer une fonction de moyenne

```
def distance_moyenne(nombre_mesures):
    somme = 0
    for i in range(nombre_mesures):
        somme = somme + capteur_ultrason.distance()
    return somme / nombre_mesures
```

Puis :

```
distance = distance_moyenne(5)
```

55. Attention à la vitesse

Faire plusieurs mesures prend du temps.

Si le robot avance rapidement, utiliser une moyenne très longue peut ralentir sa réaction.

Il faut donc chercher un équilibre entre :

```
stabilité de la mesure
```

et :

```
rapidité de réaction
```

56. Mini défi 1

Créer un programme qui :

```
mesure la distance
affiche la distance
fait un son si la distance est inférieure à 200 mm
```

57. Mini défi 2

Créer un robot qui :

```
avance à 300 mm/s
ralentit à 100 mm/s sous 400 mm
s'arrête sous 150 mm
```

58. Mini défi 3

Créer un robot qui reste approximativement à :

```
400 mm
```

d'un objet.

Si l'objet s'éloigne, le robot avance.

Si l'objet se rapproche, le robot recule.

59. Mini défi 4

Créer une fonction :

```
def obstacle_proche(limite):
```

qui retourne :

```
True
```

si un obstacle est plus proche que la limite.

Solution :

```
def obstacle_proche(limite):

    distance = capteur_ultrason.distance()

    return distance < limite
```

Puis :

```
if obstacle_proche(200):
    robot.stop()
```

60. Défi final : robot explorateur

Créer un robot capable de se déplacer automatiquement dans une zone avec plusieurs obstacles.

Le comportement doit être :

```
MESURER
↓
Obstacle loin
↓
AVANCER RAPIDEMENT
↓
Obstacle moyen
↓
RALENTIR
↓
Obstacle proche
↓
STOP
↓
RECULER
↓
TOURNER
↓
REPARTIR
```

Contraintes :

1. utiliser au moins 3 niveaux de distance
2. utiliser au moins une fonction
3. utiliser `while`
4. utiliser `if`, `elif`, `else`
5. compter le nombre d'obstacles
6. afficher chaque distance détectée
7. arrêter le programme après 5 obstacles

61. Exemple de solution

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor, UltrasonicSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase
from pybricks.tools import wait


ev3 = EV3Brick()

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_ultrason = UltrasonicSensor(Port.S4)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

def mesurer_distance():

    return capteur_ultrason.distance()

compteur = 0

while compteur < 5:

    distance = mesurer_distance()

    print(distance)

    if distance < 150:

        robot.stop()

        ev3.speaker.beep()

        compteur = compteur + 1

        robot.straight(-150)

        robot.turn(90)

    elif distance < 400:

        robot.drive(100, 0)

    else:

        robot.drive(300, 0)

    wait(20)

robot.stop()

for i in range(3):
    ev3.speaker.beep()
```

62. Questions de fin de cours

Question 1

Que mesure le capteur ultrason ?

Question 2

Quelle unité utilise :

```
capteur_ultrason.distance()
```

Question 3

Que signifie :

```
distance < 200
```

Question 4

Pourquoi le capteur ultrason permet-il d'éviter un obstacle mieux que le capteur tactile ?

Question 5

Pourquoi faut il utiliser une boucle pour surveiller la distance ?

Question 6

Pourquoi un robot rapide peut il dépasser légèrement la distance d'arrêt ?

Question 7

Qu'est ce qu'une distance cible ?

Question 8

Qu'est ce qu'une erreur ?

Question 9

Pourquoi peut on utiliser une moyenne de plusieurs mesures ?

Question 10

À quoi peut servir un coefficient `kp` ?

Correction

Réponse 1

Il estime la distance entre le capteur et un obstacle.

Réponse 2

Les millimètres.

Réponse 3

La distance mesurée est inférieure à 200 mm.

Réponse 4

Parce qu'il peut détecter l'obstacle avant le contact.

Réponse 5

Parce que la distance peut changer continuellement pendant le déplacement.

Réponse 6

À cause de l'inertie et du temps nécessaire pour mesurer, décider puis freiner.

Réponse 7

La distance que l'on souhaite maintenir entre le robot et un objet.

Réponse 8

La différence entre une valeur mesurée et une valeur souhaitée.

Réponse 9

Pour réduire l'influence des petites variations de mesure.

Réponse 10

À modifier la force avec laquelle le robot réagit à une erreur.

Difficulté fréquente

Confondre centimètres et millimètres.

Si tu écris :

```
if distance < 20:
```

en pensant demander :

```
20 cm
```

Mais cela représente :

```
20 mm
```

soit seulement :

```
2 cm
```

Il faut donc rappeler régulièrement :

```
20 cm = 200 mm
```

Autre difficulté fréquente

Créer des conditions dans le mauvais ordre.

Incorrect :

```
if distance < 600:
    vitesse = 300

elif distance < 150:
    vitesse = 0
```

La deuxième condition ne sera jamais atteinte pour une valeur sous 150.

Il faut commencer par la plus petite limite.

Cours 15 : Le capteur de couleur avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le rôle du capteur de couleur
2. connecter et initialiser un `ColorSensor`
3. utiliser `color()`
4. utiliser `reflection()`
5. comprendre la différence entre couleur et intensité lumineuse
6. faire réagir le robot à une couleur
7. détecter une ligne noire
8. arrêter le robot sur une zone colorée
9. utiliser plusieurs conditions avec le capteur
10. préparer le robot au suivi de ligne

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 moteurs EV3
3. 1 capteur de couleur EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. ruban adhésif noir
8. feuilles blanches
9. feuilles ou objets de différentes couleurs
10. un espace libre pour les tests

1. Introduction

Notre robot possède maintenant plusieurs moyens d'observer son environnement.

Avec le capteur tactile :

```
CONTACT
↓
True ou False
```

Avec le capteur ultrason :

```
DISTANCE
↓
Valeur en millimètres
```

Aujourd'hui, nous allons lui permettre de détecter ce qui se trouve sous ou devant lui grâce au capteur de couleur.

Nous pourrons par exemple créer ce comportement :

```
AVANCER
↓
Détecter une zone rouge
↓
STOP
```

Ou :

```
Sol blanc
↓
Avancer
↓
Ligne noire détectée
↓
Changer de direction
```

2. Le capteur de couleur

Le capteur de couleur EV3 peut être utilisé de plusieurs manières.

Il peut notamment mesurer :

```
une couleur

la lumière réfléchie

la lumière ambiante
```



Dans ce cours, nous allons surtout utiliser :

```
color()
```

et :

```
reflection()
```

3. Connexion du capteur

Pour ce cours, nous utiliserons :

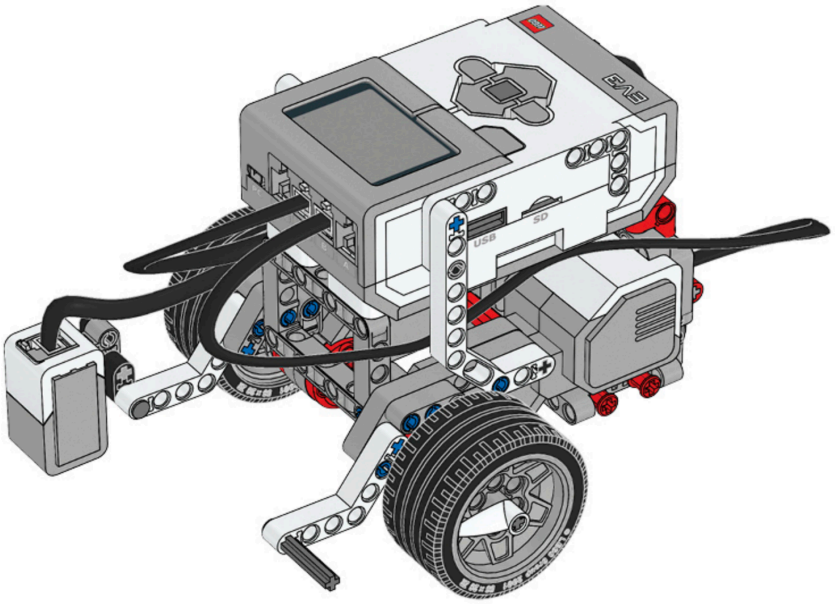
```
Port 3 = capteur de couleur
```

Notre configuration devient :

```
Port B = moteur gauche

Port C = moteur droit

Port 3 = capteur de couleur
```



4. Importer le capteur

Nous devons importer :

```
from pybricks.ev3devices import ColorSensor
```

Puis :

```
from pybricks.parameters import Port
```

Nous pouvons ensuite initialiser le capteur :

```
capteur_couleur = ColorSensor(Port.S3)
```

5. Premier programme

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port

capteur_couleur = ColorSensor(Port.S3)

print(capteur_couleur.color())
```

Placer différentes couleurs sous le capteur et relancer le programme.

6. Utiliser `Color`

Pour comparer les couleurs, nous devons également importer :

```
from pybricks.parameters import Color
```

Nous pouvons ensuite écrire :

```
if capteur_couleur.color() == Color.RED:
    print("Rouge détecté")
```

7. Exemple complet

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port, Color

capteur_couleur = ColorSensor(Port.S3)

couleur = capteur_couleur.color()

if couleur == Color.RED:
    print("Rouge")
```

8. Plusieurs couleurs

Nous pouvons utiliser `elif`.

```
couleur = capteur_couleur.color()
if couleur == Color.RED:
    print("Rouge")
elif couleur == Color.BLUE:
    print("Bleu")
elif couleur == Color.GREEN:
    print("Vert")
else:
    print("Autre couleur")
```

9. Exercice pratique 1

Créer un programme qui reconnaît :

```
rouge
bleu
vert
```

et affiche :

```
STOP
```

si la couleur rouge est détectée.

Une solution possible :

```
couleur = capteur_couleur.color()
if couleur == Color.RED:
    print("STOP")
elif couleur == Color.BLUE:
    print("BLEU")
elif couleur == Color.GREEN:
    print("VERT")
else:
    print("AUTRE")
```

10. Observer continuellement les couleurs

Nous pouvons utiliser :

```
while True:
```

Programme :

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur_couleur = ColorSensor(Port.S3)

while True:

    print(capteur_couleur.color())

    wait(200)
```

Déplacer différentes couleurs devant le capteur.

Observer les résultats.

11. Pourquoi les couleurs ne sont-elles pas toujours parfaitement reconnues ?

La détection peut être influencée par :

```
la distance entre le capteur et le sol

la lumière de la pièce

la matière de la surface

la brillance

la couleur réelle

les ombres
```

Il faut donc toujours tester le robot dans les conditions réelles.

12. Position du capteur

Pour détecter le sol, le capteur doit être placé suffisamment proche de la surface.

Il doit également rester à une hauteur relativement constante.

Si la distance entre le capteur et le sol change beaucoup, les mesures peuvent devenir moins fiables.

13. Application au robot

Initialisons le robot.

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port, Color
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

14. Avancer jusqu'au rouge

Nous voulons :

```
AVANCER

↓

SURVEILLER LE SOL

↓

ROUGE

↓

STOP
```

Programme :

```
robot.drive(200, 0)

while capteur_couleur.color() != Color.RED:
    pass

robot.stop()
```

15. Comprendre la condition

```
while capteur_couleur.color() != Color.RED:
```

signifie :

```
TANT QUE

la couleur n’est pas rouge
```

le robot continue.

Lorsque le rouge est détecté, la condition devient fausse.

Le programme sort de la boucle.

16. Exercice pratique 2

Placer une bande rouge au sol.

Le robot doit avancer jusqu’à cette bande puis s’arrêter.

Solution :

```
robot.drive(200, 0)

while capteur_couleur.color() != Color.RED:
    pass

robot.stop()
```

17. Ajouter une réaction selon la couleur

Nous pouvons créer plusieurs comportements.

Par exemple :

```
VERT

↓

AVANCER
JAUNE

↓

RALENTIR
ROUGE

↓

STOP
```

18. Programme avec plusieurs couleurs

```
while True:
    couleur = capteur_couleur.color()
    if couleur == Color.RED:
        robot.stop()
    elif couleur == Color.YELLOW:
        robot.drive(100, 0)
    elif couleur == Color.GREEN:
        robot.drive(300, 0)
```

19. Ajouter un comportement par défaut

Il vaut mieux prévoir ce que le robot doit faire lorsqu'aucune des couleurs attendues n'est détectée.

```
while True:
    couleur = capteur_couleur.color()
    if couleur == Color.RED:
        robot.stop()
    elif couleur == Color.YELLOW:
        robot.drive(100, 0)
    elif couleur == Color.GREEN:
        robot.drive(300, 0)
    else:
        robot.drive(200, 0)
```

20. Défi pratique 1

Créer une piste avec trois zones.

```
VERT
JAUNE
ROUGE
```

Le robot doit :

```
VERT
↓
avancer rapidement
JAUNE
↓
ralentir
ROUGE
↓
s'arrêter
```

21. La réflexion lumineuse

Reconnaître une couleur précise est utile.

Mais pour suivre une ligne, nous avons souvent besoin d'une mesure plus fine.

Nous pouvons utiliser :

```
capteur_couleur.reflection()
```

Cette méthode mesure la quantité de lumière réfléchie par la surface.

22. Valeur de réflexion

La réflexion est généralement exprimée sur une échelle allant approximativement de :

```
0 à 100
```

Une surface sombre renvoie peu de lumière.

Une surface claire renvoie davantage de lumière.

On peut donc avoir approximativement :

```
NOIR
↓
10
```

et :

```
BLANC
↓
80
```

Les valeurs exactes dépendent du terrain et du capteur.

23. Premier programme avec `reflection()`

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur_couleur = ColorSensor(Port.S3)

while True:

    valeur = capteur_couleur.reflection()

    print(valeur)

    wait(200)
```

Déplacer le capteur entre une zone noire et une zone blanche.

24. Expérience de calibration

Mesurer plusieurs fois le noir.

Noter :

Mesure noir	Valeur
1	
2	
3	
4	
5	

Puis mesurer le blanc.

Mesure blanc	Valeur
1	
2	
3	
4	
5	

25. Calculer une moyenne

Supposons que nous obtenions :

```
Noir ≈ 12
```

```
Blanc ≈ 78
```

Nous pouvons choisir une valeur centrale.

```
(12 + 78) / 2
```

donne :

```
45
```

Nous pouvons donc utiliser :

```
seuil = 45
```

26. Qu’est-ce qu’un seuil ?

Un seuil permet de séparer deux catégories.

Par exemple :

```
reflection < 45
```

```
↓
```

```
surface sombre
```

et :

```
reflection >= 45
```

```
↓
```

```
surface claire
```

27. Premier détecteur noir ou blanc

```
seuil = 45
valeur = capteur_couleur.reflection()
if valeur < seuil:
    print("NOIR")
else:
    print("BLANC")
```

28. Exercice pratique 3

Mesurer le noir et le blanc de ton terrain.

Définir ensuite :

```
valeur_noir = ...
valeur_blanc = ...
```

Puis calculer :

```
seuil = (valeur_noir + valeur_blanc) / 2
```

Afficher le résultat.

29. Programme automatique

```
valeur_noir = 12
valeur_blanc = 78

seuil = (valeur_noir + valeur_blanc) / 2

print(seuil)
```

Résultat :

```
45
```

30. Avancer jusqu'à une ligne noire

Nous pouvons maintenant créer :

```
AVANCER
↓
SOL BLANC
↓
CONTINUER
↓
LIGNE NOIRE
↓
STOP
```

Programme :

```
seuil = 45

robot.drive(200, 0)

while capteur_couleur.reflection() >= seuil:
    pass

robot.stop()
```

31. Pourquoi utiliser `reflection()` plutôt que `color()` ?

Nous pourrions essayer :

```
Color.BLACK
```

Mais `reflection()` nous donne une valeur numérique.

Cela permet :

```
une calibration plus précise

un réglage du seuil

des corrections progressives

le suivi de ligne
```

La réflexion sera donc très importante pour les prochains cours.

32. Exercice pratique 4

Créer un robot qui avance sur une zone blanche et s'arrête dès qu'il rencontre une ligne noire.

Faire trois essais avec différentes vitesses.

```
100 mm/s

200 mm/s

400 mm/s
```

Noter la précision.

Vitesse	Dépassement de la ligne
100	
200	
400	

33. Pourquoi le robot dépasse t-il la ligne ?

Comme avec le capteur ultrason, le robot possède une inertie.

Le temps nécessaire pour :

```
mesurer

décider

freiner
```

provoque un petit dépassement.

Plus le robot avance rapidement, plus ce dépassement peut être important.

34. Utiliser une fonction

```
def est_noir():

    valeur = capteur_couleur.reflection()

    return valeur < seuil
```

Nous pouvons ensuite écrire :

```
if est_noir():
    robot.stop()
```

Ou :

```
while not est_noir():
    robot.drive(200, 0)
```

35. Fonction de calibration simple

Nous pouvons créer :

```
def calcul_seuil(noir, blanc):

    return (noir + blanc) / 2
```

Puis :

```
seuil = calcul_seuil(12, 78)
```

36. Pourquoi calibrer ?

Supposons que dans une autre salle nous mesurions :

```
Noir = 20
```

```
Blanc = 60
```

Le seuil idéal devient :

```
40
```

et non :

```
45
```

Les valeurs dépendent donc de l'environnement.

37. Première logique de suivi de ligne

Imaginons une ligne noire sur un fond blanc.

Nous pouvons écrire :

```
while True:
    valeur = capteur_couleur.reflection()
    if valeur < seuil:
        robot.drive(150, 50)
    else:
        robot.drive(150, -50)
```

Le robot tourne d'un côté sur le noir et de l'autre côté sur le blanc.

38. Que va faire le robot ?

Il va approximativement :

```
détecter noir
↓
tourner d'un côté
↓
détecter blanc
↓
tourner dans l'autre sens
↓
détecter noir
↓
recommencer
```

Le robot va donc zigzaguer autour du bord de la ligne.

C'est notre premier suivi de ligne.

39. Pourquoi suivre le bord de la ligne ?

Si le capteur était parfaitement au centre d'une large ligne noire, nous ne saurions pas dans quel sens tourner lorsqu'il quitte la ligne.

Il est donc souvent plus intéressant de suivre la frontière entre :

```
NOIR
```

et :

```
BLANC
```

Cette frontière correspond approximativement à notre valeur de seuil.

40. Schéma du suivi de bord

```
BLANC            NOIR
```



```
        ↑

      capteur

   valeur cible
```

Le robot essaie de rester proche de la transition entre les deux surfaces.

41. Premier programme complet de suivi

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

valeur_noir = 12
valeur_blanc = 78
seuil = (valeur_noir + valeur_blanc) / 2

while True:

    valeur = capteur_couleur.reflection()

    if valeur < seuil:
        robot.drive(150, 60)
    else:
        robot.drive(150, -60)
```

42. Tester le sens

Selon la position du capteur et le côté de la ligne suivi, il faudra peut être inverser :

```
60
```

et :

```
moins 60
```

Il faut observer le comportement du robot.

43. Un problème apparaît

Le robot va souvent zigzaguer fortement.

Pourquoi ?

Parce que nous utilisons seulement deux choix.

```
NOIR
↓
tourner fort à gauche
```

ou :

```
BLANC
↓
tourner fort à droite
```

Il n’existe aucune correction intermédiaire.

44. Vers une correction progressive

Imaginons :

```
seuil = 45
```

Si le capteur mesure :

```
44
```

nous sommes très proches de la valeur souhaitée.

Mais avec notre programme actuel, nous tournons aussi fortement que si la valeur était :

```
10
```

Ce n’est pas idéal.

Nous avons besoin de calculer une erreur.

45. Calculer l’erreur

Nous pouvons écrire :

```
erreur = seuil - valeur
```

Exemple :

```
seuil = 45

valeur = 40

erreur = 5
```

Autre exemple :

```
seuil = 45

valeur = 20

erreur = 25
```

Dans le deuxième cas, nous sommes beaucoup plus loin de la valeur cible.

Nous pourrions donc tourner davantage.

46. Une correction proportionnelle

Nous pouvons calculer :

```
correction = erreur * kp
```

Par exemple :

```
kp = 2
```

Si :

```
erreur = 5
```

alors :

```
correction = 10
```

Si :

```
erreur = 25
```

alors :

```
correction = 50
```

Le robot corrige donc davantage lorsque l’erreur est importante.

47. Première version proportionnelle

```
seuil = 45

kp = 2

vitesse = 150

while True:

    valeur = capteur_couleur.reflection()

    erreur = seuil - valeur

    correction = erreur * kp

    robot.drive(vitesse, correction)
```

Ce programme peut permettre un suivi beaucoup plus fluide.

Nous étudierons cette méthode plus en détail dans les cours suivants.

48. Visualiser les valeurs

Pendant les tests, nous pouvons afficher :

```
print(valeur)

print(erreur)

print(correction)
```

Cela permet de comprendre ce que fait le contrôleur.

49. Exercice pratique 5

Avec :

```
seuil = 50

kp = 2
```

calculer la correction pour :

```
valeur = 40
```

Calcul :

```
erreur = 50 moins 40 = 10
```

Puis :

```
correction = 10 × 2 = 20
```

50. Deuxième exemple

Avec :

```
seuil = 50
```

```
valeur = 70
```

```
kp = 2
```

nous avons :

```
erreur = moins 20
```

Donc :

```
correction = moins 40
```

Le signe indique le sens de la correction.

51. Couleur ou réflexion ?

Nous avons maintenant deux façons d'utiliser le capteur.

`color()`

Utilisé lorsqu'on veut reconnaître une couleur.

Exemple :

```
if capteur_couleur.color() == Color.RED:
```

Très utile pour :

```
zones de commande
```

```
marqueurs
```

```
arrêt sur une couleur
```

`reflection()`

Utilisé lorsqu'on veut mesurer la luminosité réfléchie.

Exemple :

```
valeur = capteur_couleur.reflection()
```

Très utile pour :

```
détection noir et blanc
```

```
seuils
```

```
suivi de ligne
```

```
régulation
```

52. Défi pratique 6 : feux de circulation

Créer trois zones au sol :

```
VERT

JAUNE

ROUGE
```

Le robot doit :

```
VERT

↓

300 mm/s
JAUNE

↓

100 mm/s
ROUGE

↓

STOP
```

53. Défi pratique 7 : arrêt sur ligne

Le robot doit :

```
avancer pendant 500 mm

continuer ensuite jusqu'à trouver une ligne noire

s'arrêter
```

Une solution :

```
robot.straight(500)

robot.drive(150, 0)

while capteur_couleur.reflection() >= seuil:
    pass

robot.stop()
```

54. Défi pratique 8 : compter les lignes

Créer plusieurs bandes noires sur fond blanc.

Le robot doit compter combien de lignes il traverse.

Il faut éviter de compter plusieurs fois la même ligne.

55. Première idée

Nous pouvons utiliser :

```
compteur = 0

while True:

    if capteur_couleur.reflection() < seuil:

        compteur = compteur + 1

        print(compteur)
```

Mais ce programme possède un problème.

Tant que le capteur reste sur la bande noire, le compteur continue d'augmenter.

56. Attendre de quitter la ligne

Nous pouvons écrire :

```
compteur = 0

while True:

    if capteur_couleur.reflection() < seuil:

        compteur = compteur + 1

        print(compteur)

        while capteur_couleur.reflection() < seuil:
            pass
```

Le programme attend que le robot quitte la bande noire avant de pouvoir compter une nouvelle ligne.

57. Défi pratique 9

Faire avancer le robot et l'arrêter après :

```
3 lignes noires
```

Une solution possible :

```
compteur = 0
robot.drive(150, 0)

while compteur < 3:

    if capteur_couleur.reflection() < seuil:

        compteur = compteur + 1

        while capteur_couleur.reflection() < seuil:
            pass

robot.stop()
```

58. Défi final : parcours intelligent

Créer un parcours contenant :

```
une zone verte

une ligne noire

une zone jaune

une deuxième ligne noire

une zone rouge
```

Le robot doit :

```
démarrer
↓
sur vert, avancer rapidement
↓
compter la première ligne noire
↓
sur jaune, ralentir
↓
compter la deuxième ligne noire
↓
sur rouge, s'arrêter définitivement
```

Contraintes :

1. utiliser `ColorSensor`
2. utiliser `color()`
3. utiliser `reflection()`
4. utiliser au moins une fonction
5. utiliser une boucle
6. utiliser une condition
7. compter les lignes détectées

59. Exemple de solution

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port, Color
from pybricks.robotics import DriveBase
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

valeur_noir = 12
valeur_blanc = 78

seuil = (valeur_noir + valeur_blanc) / 2

compteur_lignes = 0

termine = False

while not termine:

    couleur = capteur_couleur.color()

    valeur = capteur_couleur.reflection()

    if couleur == Color.RED:

        robot.stop()

        termine = True

    elif couleur == Color.YELLOW:

        robot.drive(100, 0)

    elif couleur == Color.GREEN:

        robot.drive(300, 0)

    else:

        robot.drive(200, 0)

    if valeur < seuil:

        compteur_lignes = compteur_lignes + 1

        print(compteur_lignes)

        while capteur_couleur.reflection() < seuil:
            wait(10)

robot.stop()
```

60. Questions de fin de cours

Question 1

À quoi sert le capteur de couleur ?

Question 2

Quelle méthode permet de détecter une couleur ?

Question 3

Quelle méthode permet de mesurer la lumière réfléchie ?

Question 4

Pourquoi faut il calibrer le noir et le blanc ?

Question 5

Qu’est ce qu’un seuil ?

Question 6

Pourquoi `reflection()` est elle particulièrement intéressante pour le suivi de ligne ?

Question 7

Pourquoi faut il parfois attendre de quitter une bande noire avant d’incrémenter un compteur ?

Question 8

Qu’est ce qu’une valeur cible ?

Question 9

Qu’est ce qu’une erreur dans un système de régulation ?

Question 10

À quoi sert `kp` ?

Correction

Réponse 1

À détecter des couleurs et à mesurer la quantité de lumière réfléchie par une surface.

Réponse 2

```
color()
```

Réponse 3

```
reflection()
```

Réponse 4

Parce que les valeurs dépendent du sol, de la lumière et de la position du capteur.

Réponse 5

Une valeur permettant de séparer deux catégories de mesures.

Réponse 6

Parce qu'elle fournit une valeur numérique qui permet de mesurer à quel point le capteur est proche du noir ou du blanc.

Réponse 7

Pour éviter de compter plusieurs fois la même bande.

Réponse 8

La valeur que le système essaie de maintenir.

Réponse 9

La différence entre la valeur souhaitée et la valeur mesurée.

Réponse 10

À déterminer la force de la correction proportionnelle.

Difficulté fréquente

Donner directement :

```
seuil = 50
```

sans calibration.

Cette valeur peut fonctionner sur un terrain et être mauvaise sur un autre.

Tu dois apprendre à mesurer avant de régler.

Autre difficulté fréquente

Confondre :

```
capteur_couleur.color()
```

avec :

```
capteur_couleur.reflection()
```

Il faut rappeler :

```
color()
```

```
↓
```

```
nom d'une couleur
```

et :

```
reflection()
```

```
↓
```

```
valeur numérique
```

Cours 16 : Détection de ligne et suivi de ligne simple avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le principe du suivi de ligne
2. calibrer le capteur sur une surface noire et blanche
3. calculer une valeur seuil
4. faire la différence entre suivre le centre et suivre le bord d'une ligne
5. utiliser `reflection()` pour corriger la direction du robot
6. créer un suivi de ligne simple avec `if` et `else`
7. régler la vitesse du robot
8. régler la force de rotation
9. identifier les causes d'un suivi instable
10. améliorer progressivement le comportement du robot

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur de couleur EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. une piste blanche avec une ligne noire
8. du ruban adhésif noir
9. une règle
10. un espace libre pour les tests

1. Introduction

Lors du cours précédent, nous avons appris à utiliser :

```
capteur_couleur.reflection()
```

pour mesurer la quantité de lumière réfléchie.

Nous avons par exemple obtenu :

```
Noir ≈ 12
Blanc ≈ 78
```

Puis nous avons calculé :

```
seuil = (12 + 78) / 2
```

Ce qui donne :

```
45
```

Aujourd'hui, nous allons utiliser cette valeur pour permettre au robot de suivre automatiquement une ligne.

2. Le problème

Nous avons une piste composée d’un fond blanc et d’une ligne noire.

```
BLANC        NOIR        BLANC
```



Nous voulons que le robot avance en suivant la ligne sans intervention humaine.

Le robot doit donc continuellement :

```
MESURER
↓
DÉCIDER
↓
CORRIGER
↓
MESURER À NOUVEAU
```

3. Où placer le capteur ?

Le capteur de couleur est généralement placé près du sol.

Il doit regarder la surface juste devant ou sous le robot.

Il faut éviter qu’il soit :

```
trop haut

trop bas

incliné

mobile
```

La hauteur doit rester aussi constante que possible.

4. Faut-il suivre le centre de la ligne ?

Imaginons que le capteur soit complètement au centre d’une large ligne noire.

Il mesure :

```
NOIR
```

Mais si le robot quitte légèrement la ligne, comment savoir de quel côté il est sorti ?

Le robot peut difficilement déterminer s'il doit tourner :

```
à gauche
```

ou :

```
à droite
```

avec un seul capteur.

5. Suivre le bord de la ligne

Une solution courante consiste à suivre la frontière entre :

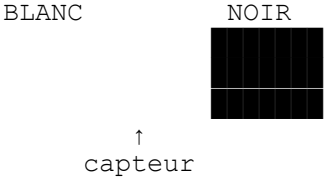
```
NOIR
```

et :

```
BLANC
```

On parle de suivi du bord de ligne.

On peut représenter cela ainsi :



Le capteur essaie de rester proche de la transition entre les deux surfaces.

6. Calibration

Avant de commencer, nous devons mesurer la valeur du noir et du blanc.

Programme :

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur_couleur = ColorSensor(Port.S3)

while True:

    print(capteur_couleur.reflection())

    wait(200)
```

Placer le capteur sur le noir puis sur le blanc.

7. Relever les mesures

Exemple :

Surface	Mesure 1	Mesure 2	Mesure 3	Moyenne
Noir	12	13	11	12
Blanc	77	79	78	78

Nous obtenons :

```
valeur_noir = 12
valeur_blanc = 78
```

8. Calcul du seuil

Nous calculons :

```
seuil = (valeur_noir + valeur_blanc) / 2
```

Donc :

```
(12 + 78) / 2

=

45
```

Notre valeur cible est donc environ :

```
45
```

9. Premier comportement

Nous allons commencer avec une logique très simple.

Si le capteur voit une surface sombre :

```
tourner à gauche
```

Sinon :

```
tourner à droite
```

En Python :

```
if valeur < seuil:
    tourner à gauche
else:
    tourner à droite
```

10. Utiliser `DriveBase`

Initialisons le robot.

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

11. Premier suivi de ligne

```
valeur_noir = 12
valeur_blanc = 78

seuil = (valeur_noir + valeur_blanc) / 2

while True:

    valeur = capteur_couleur.reflection()

    if valeur < seuil:

        robot.drive(150, 70)

    else:

        robot.drive(150, -70)
```

Le robot doit commencer à suivre approximativement le bord de la ligne.

12. Comprendre `robot.drive()`

Nous utilisons :

```
robot.drive(vitesse, rotation)
```

Le premier paramètre correspond à la vitesse d'avancement.

Par exemple :
`robot.drive(150, 70)`

signifie approximativement :

```
avancer à 150 mm/s
avec une rotation de 70°/s
```

Le robot suit donc une courbe.

13. Que se passe t-il sur le noir ?

Supposons :

```
seuil = 45
```

```
valeur = 20
```

Nous avons :

```
20 < 45
```

La condition est vraie.

Le programme exécute :

```
robot.drive(150, 70)
```

Le robot tourne dans un sens.

14. Que se passe t-il sur le blanc ?

Supposons :

```
valeur = 70
```

Nous avons :

```
70 < 45
```

La condition est fausse.

Le programme exécute :

```
robot.drive(150, -70)
```

Le robot tourne dans l'autre sens.

15. Le mouvement obtenu

Le robot effectue approximativement :

```
NOIR
↓
tourner dans un sens
↓
BLANC
↓
tourner dans l'autre sens
↓
NOIR
↓
recommencer
```

Il zigzague autour de la frontière.

16. Pourquoi le robot zigzague t il ?

Notre programme possède uniquement deux niveaux de correction :

```
plus 70
```

ou :

```
moins 70
```

Même si le robot est seulement légèrement trop loin du bord, il effectue la même correction que s'il était très loin.

Le mouvement est donc relativement brutal.

17. Exercice pratique 1

Tester :

```
robot.drive(150, 30)
```

et :

```
robot.drive(150, -30)
```

Puis tester :

```
plus ou moins 70
```

Puis :

```
plus ou moins 120
```

Compléter :

Rotation	Comportement
30	
70	
120	

18. Que représente la force de rotation ?

Plus la deuxième valeur est importante, plus le robot change rapidement de direction.

Avec :

```
robot.drive(150, 20)
```

le virage est doux.

Avec :

```
robot.drive(150, 150)
```

le virage est beaucoup plus important.

19. Trouver un compromis

Si la correction est trop faible :

```
le robot réagit trop lentement

il peut perdre la ligne
```

Si elle est trop forte :

```
le robot zigzague beaucoup

le mouvement devient instable
```

Nous devons trouver un compromis.

20. Effet de la vitesse

La vitesse d'avancement influence également le comportement.

Tester :

```
vitesse = 80
```

puis :

```
vitesse = 150
```

puis :

```
vitesse = 300
```

21. Exercice pratique 2

Utiliser :
`correction = 70`
et tester :

Vitesse	Résultat
80 mm/s	
150 mm/s	
300 mm/s	

Question :

Quelle vitesse permet de suivre la ligne de manière fiable ?

22. Pourquoi une vitesse élevée est-elle plus difficile ?

À grande vitesse :

```
le robot parcourt davantage de distance entre deux corrections

l’inertie devient plus importante

les erreurs augmentent plus rapidement
```

Il faut donc souvent augmenter la qualité du contrôleur avant d’augmenter la vitesse.

23. Le sens de correction

Selon le montage et le côté de la ligne suivi, ce programme peut tourner dans le mauvais sens.

```
if valeur < seuil:
    robot.drive(150, 70)
else:
    robot.drive(150, -70)
```

Dans ce cas, il suffit d’inverser :

```
if valeur < seuil:
    robot.drive(150, -70)
else:
    robot.drive(150, 70)
```

24. Comment savoir quel sens utiliser ?

Placer le robot sur le bord de la ligne.

Déplacer manuellement le capteur vers le noir.

Demander :

Dans quel sens le robot doit il tourner pour revenir vers le bord ?

Faire la même chose vers le blanc.

Tu dois comprendre la correction plutôt que mémoriser les signes.

25. Fonction de calibration

Nous pouvons créer :

```
def calcul_seuil(noir, blanc):
    return (noir + blanc) / 2
```

Puis :

```
seuil = calcul_seuil(12, 78)
```

26. Créer une fonction de suivi

```
def suivre_ligne():
    while True:
        valeur = capteur_couleur.reflection()
        if valeur < seuil:
            robot.drive(150, 70)
        else:
            robot.drive(150, -70)
```

Puis :

```
suivre_ligne()
```

27. Paramétrer la fonction

Encore mieux :

```
def suivre_ligne(vitesse, correction):
    while True:
        valeur = capteur_couleur.reflection()
        if valeur < seuil:
            robot.drive(vitesse, correction)
        else:
            robot.drive(vitesse, -correction)
```

Nous pouvons tester :

```
suivre_ligne(100, 50)
```

ou :

```
suivre_ligne(200, 90)
```

28. Problème de la boucle infinie

Avec :

```
while True:
```

le suivi continue indéfiniment.

Mais nous voudrons parfois suivre la ligne seulement :

```
pendant une certaine distance

jusqu'à une couleur

jusqu'à un obstacle

pendant une certaine durée
```

Nous devons donc prévoir des conditions d'arrêt.

29. Suivre la ligne pendant une distance

Nous pouvons utiliser :

```
robot.reset()
```

puis surveiller :

```
robot.distance()
```

Exemple :

```
robot.reset()
while robot.distance() < 1000:
    valeur = capteur_couleur.reflection()
    if valeur < seuil:
        robot.drive(150, 70)
    else:
        robot.drive(150, -70)
robot.stop()
```

Le robot suit la ligne pendant environ :

```
1000 mm
```

30. Exercice pratique 3

Faire suivre la ligne pendant :

```
2 mètres
```

Conversion :

```
2 m = 2000 mm
```

Il faut donc utiliser :

```
while robot.distance() < 2000:
```

31. Fonction avec une distance

```
def suivre_ligne_distance(distance):
    robot.reset()
    while robot.distance() < distance:
        valeur = capteur_couleur.reflection()
        if valeur < seuil:
            robot.drive(150, 70)
        else:
            robot.drive(150, -70)
    robot.stop()
```

Puis :

```
suivre_ligne_distance(1000)
```

32. Suivre jusqu'à une couleur

Nous pouvons également utiliser la reconnaissance des couleurs.

Par exemple :

```
suivre la ligne
jusqu'à une zone rouge
puis arrêter
```

Une idée simple :

```
while capteur_couleur.color() != Color.RED:
    # suivi de ligne
```

33. Attention

Selon la piste, alterner rapidement entre `color()` et `reflection()` peut donner un comportement moins prévisible qu'un usage simple de `reflection()`.

Pour les exercices de base, il est souvent préférable d'utiliser un second marqueur bien visible ou de tester soigneusement le comportement.

L'objectif de cette leçon reste surtout le suivi de ligne simple.

34. Suivre jusqu’à une ligne transversale

Nous pouvons aussi créer une large bande noire traversant la piste.

Mais notre capteur voit déjà régulièrement du noir pendant le suivi.

Nous ne pouvons donc pas simplement écrire :

```
if valeur < seuil:
    stop
```

Il faut une stratégie différente.

35. Utiliser un deuxième événement

Par exemple :

```
une zone rouge
```

est plus simple à reconnaître.

Ou utiliser :

```
un capteur tactile

un capteur ultrason

une distance parcourue
```

Nous apprendrons progressivement à combiner plusieurs capteurs.

36. Afficher la valeur pendant le suivi

Pendant le débogage :

```
while True:

    valeur = capteur_couleur.reflection()

    print(valeur)

    if valeur < seuil:

        robot.drive(150, 70)

    else:

        robot.drive(150, -70)
```

Cela permet d’observer les valeurs rencontrées.

37. Afficher le côté détecté

```
if valeur < seuil:
    print("NOIR")
    robot.drive(150, 70)
else:
    print("BLANC")
    robot.drive(150, -70)
```

Cela peut être utile pour comprendre un comportement incorrect.

38. Utiliser une variable pour le sens

Nous pouvons définir :

```
sens = 1
```

Puis :

```
if valeur < seuil:
    robot.drive(vitesse, correction * sens)
else:
    robot.drive(vitesse, -correction * sens)
```

Si nous voulons suivre l'autre côté de la ligne :

```
sens = -1
```

39. Fonction plus réutilisable

```
def suivre_ligne(vitesse, correction, sens=1):
    while True:
        valeur = capteur_couleur.reflection()
        if valeur < seuil:
            robot.drive(
                vitesse,
                correction * sens
            )
        else:
            robot.drive(
                vitesse,
                -correction * sens
            )
```

Puis :

```
suivre_ligne(150, 70, 1)
```

ou :

```
suivre_ligne(150, 70, -1)
```

40. Exercice pratique 4

Faire suivre successivement :

```
le bord gauche de la ligne
```

puis :

```
le bord droit de la ligne
```

en modifiant uniquement :

```
sens
```

41. Mesurer la qualité du suivi

Dire simplement :

```
ça fonctionne
```

n’est pas très précis.

Nous pouvons définir plusieurs critères :

```
nombre de sorties de piste
temps pour parcourir un tour
distance moyenne du bord
importance des oscillations
```

42. Petit protocole de test

Faire trois tours avec chaque réglage.

Par exemple :

Vitesse	Correction	Sorties de piste	Temps
100	50		
150	70		
200	90		

Tu peux alors comparer les réglages.

43. Introduction à la fréquence de boucle

Notre boucle effectue continuellement :

```
mesure

correction

mesure

correction
```

Plus elle se répète rapidement, plus le robot peut réagir rapidement.

Il ne faut donc pas ajouter de grandes attentes comme :

```
wait(500)
```

dans le suivi de ligne.

Sinon le robot attendrait une demi seconde entre deux corrections.

44. Expérience avec `wait()`

Tester volontairement :

```
wait(500)
```

dans la boucle.

Observer le résultat.

Puis essayer :

```
wait(50)
```

Puis sans attente.

Tu peux constater l'importance de la fréquence de correction.

45. Exercice de réflexion

Pourquoi ce programme risque t-il de mal fonctionner ?

```
while True:

    valeur = capteur_couleur.reflection()

    if valeur < seuil:
        robot.drive(150, 70)

    else:
        robot.drive(150, -70)

    wait(1000)
```

Réponse :

Le robot attend une seconde avant de recalculer sa direction.

Il parcourt donc une grande distance sans nouvelle correction.

46. Première limite de notre contrôleur

Imaginons :

```
seuil = 45
```

Cas A :

```
valeur = 44
```

Nous sommes presque exactement sur la frontière.

Pourtant, notre programme applique :

```
correction = 70
```

Cas B :

```
valeur = 10
```

Nous sommes très loin de la frontière.

Le programme applique encore :

```
correction = 70
```

Le robot ne fait donc aucune différence entre une petite erreur et une grande erreur.

47. Une meilleure idée

Nous pouvons calculer :

```
erreur = seuil - valeur
```

Puis :

```
correction = erreur * kp
```

Ainsi :

```
petite erreur
↓
petite correction
```

et :

```
grande erreur
↓
grande correction
```

48. Exemple

Avec :

```
seuil = 45
```

```
kp = 2
```

Si :

```
valeur = 43
```

alors :

```
erreur = 2
```

```
correction = 4
```

Si :

```
valeur = 20
```

alors :

```
erreur = 25
```

```
correction = 50
```

Le deuxième cas reçoit une correction beaucoup plus importante.

49. Comparaison

Contrôleur simple

```
if valeur < seuil:
    correction = 70
else:
    correction = -70
```

La correction possède seulement deux valeurs.

Contrôleur proportionnel

```
erreur = seuil - valeur
```

```
correction = erreur * kp
```

La correction peut prendre beaucoup de valeurs.

Par exemple :

```
4

12

25

50

moins 10

moins 40
```

50. Premier aperçu du prochain cours

Le programme proportionnel ressemblera à :

```
vitesse = 150
kp = 2

while True:

    valeur = capteur_couleur.reflection()

    erreur = seuil - valeur

    correction = erreur * kp

    robot.drive(vitesse, correction)
```

Nous allons étudier précisément ce programme lors de la prochaine leçon.

51. Défi pratique 5 : suivi lent

Créer un suivi de ligne simple avec :

```
vitesse = 100 mm/s
```

Tu dois trouver par toi-même une bonne valeur pour :

```
correction
```

52. Défi pratique 6 : suivi rapide

Augmenter progressivement :

```
100

150

200

250 mm/s
```

À chaque vitesse, adapter la correction.

Noter :

Vitesse	Correction choisie	Stable ?
100		
150		
200		
250		

53. Défi pratique 7 : parcours complet

Créer une piste comprenant :

```
une ligne droite

une courbe légère

une courbe serrée

une deuxième ligne droite
```

Le robot doit effectuer un tour complet sans quitter la piste.

54. Défi pratique 8 : trois tours

Modifier le programme pour suivre la ligne pendant une distance correspondant approximativement à trois tours.

Ou, si la longueur d'un tour est connue :

```
distance_un_tour = 3000
distance_totale = distance_un_tour * 3
```

Puis :

```
suivre_ligne_distance(distance_totale)
```

55. Défi final : optimisation

Chaque étudiant dispose de la même piste.

Objectif :

```
réaliser un tour complet
sans sortir de la piste
le plus rapidement possible
```

Tu peux modifier uniquement :

```
Vitesse
```
et :
```
correction
```

Tu dois noter chaque essai.

56. Tableau d’optimisation

Essai	Vitesse	Correction	Temps	Sortie de ligne
1				
2				
3				
4				
5				

Le meilleur réglage doit être justifié.

57. Exemple de solution complète

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

valeur_noir = 12
valeur_blanc = 78

seuil = (valeur_noir + valeur_blanc) / 2

vitesse = 150
correction = 70

robot.reset()

while robot.distance() < 2000:

    valeur = capteur_couleur.reflection()

    if valeur < seuil:

        robot.drive(
            vitesse,
            correction
        )

    else:

        robot.drive(
            vitesse,
            -correction
        )

robot.stop()
```

58. Questions de fin de cours

Question 1

Pourquoi est il plus simple de suivre le bord d'une ligne avec un seul capteur ?

Question 2

Comment calcule t on généralement le seuil entre noir et blanc ?

Question 3

Que fait :

```
capteur_couleur.reflection()
```

Question 4

Pourquoi utilise t on une boucle pendant le suivi de ligne ?

Question 5

Que représente le premier paramètre de :

```
robot.drive(150, 70)
```

Question 6

Que représente le deuxième paramètre ?

Question 7

Pourquoi une correction trop faible est elle problématique ?

Question 8

Pourquoi une correction trop forte est elle problématique ?

Question 9

Pourquoi le suivi simple zigzague t il ?

Question 10

Quelle amélioration allons nous utiliser au prochain cours ?

Correction

Réponse 1

Parce que la frontière entre noir et blanc fournit une valeur cible claire permettant de savoir de quel côté le robot se trouve.

Réponse 2

```
seuil = (noir + blanc) / 2
```

Réponse 3

Elle mesure la quantité de lumière réfléchie par la surface.

Réponse 4

Parce que le robot doit continuellement mesurer la surface et corriger sa direction.

Réponse 5

La vitesse d’avancement du robot en millimètres par seconde.

Réponse 6

La vitesse de rotation ou correction de direction.

Réponse 7

Le robot peut ne pas revenir assez rapidement vers la ligne.

Réponse 8

Le robot peut osciller fortement et devenir instable.

Réponse 9

Parce que le contrôleur possède seulement deux corrections fixes.

Réponse 10

Une correction proportionnelle à l’erreur.

Difficulté fréquente

Le robot part immédiatement à l'opposé de la ligne.

Dans ce cas, le sens de la correction est probablement inversé.

Il faut inverser :

```
plus correction
```

et :

```
moins correction
```

ou utiliser :

```
sens = -1
```

Difficulté fréquente

Le robot perd la ligne dans les courbes.

Solutions à tester :

1. diminuer la vitesse
2. augmenter la correction
3. vérifier le seuil
4. vérifier la hauteur du capteur
5. vérifier que la piste possède un contraste suffisant

Difficulté fréquente

Le robot zigzague énormément même sur une ligne droite.

Ce comportement est en partie normal avec ce type de contrôleur.

C'est précisément la raison pour laquelle nous allons introduire le contrôle proportionnel.

Cours 17 : Le suivi de ligne proportionnel avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre la notion d'erreur
2. calculer une erreur entre une valeur cible et une valeur mesurée
3. comprendre le principe d'une correction proportionnelle
4. utiliser un coefficient `kp`
5. faire varier la correction selon l'importance de l'erreur
6. régler un contrôleur proportionnel
7. identifier les oscillations
8. trouver un compromis entre vitesse et stabilité
9. comparer un suivi simple et un suivi proportionnel
10. préparer l'introduction au contrôleur PID

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur de couleur EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. une piste blanche avec une ligne noire
8. une ligne droite
9. plusieurs courbes
10. une piste complète pour le défi final

1. Rappel du cours précédent

Lors du cours précédent, nous avons créé un suivi de ligne simple.

```
while True:

    valeur = capteur_couleur.reflection()

    if valeur < seuil:

        robot.drive(150, 70)

    else:

        robot.drive(150, -70)
```

Le robot possède seulement deux comportements.

```
Surface sombre

↓

tourner dans un sens
Surface claire

↓

tourner dans l'autre sens
```

Cette méthode fonctionne, mais le robot zigzague beaucoup.

2. Le problème

Imaginons que notre seuil soit :

```
seuil = 45
```

Le capteur mesure :

```
44
```

Nous sommes très proches de la valeur souhaitée.

Mais notre ancien programme applique quand même une correction de :

```
70
```

Maintenant, imaginons que le capteur mesure :

```
10
```

Le robot est beaucoup plus éloigné de la frontière.

Pourtant, il applique encore :

```
70
```

Notre contrôleur ne fait donc pas la différence entre :

```
une petite erreur
```

et :

```
une grande erreur
```

3. Une meilleure stratégie

Nous voudrions obtenir :

```
Petite erreur

↓

Petite correction
```

et :

```
Grande erreur

↓

Grande correction
```

Pour cela, nous allons calculer la différence entre :

```
la valeur souhaitée
```

et :

```
la valeur mesurée
```

4. La valeur cible

Nous avons calibré le noir et le blanc.

Par exemple :

```
valeur_noir = 12
valeur_blanc = 78
```

Nous calculons :

```
cible = (valeur_noir + valeur_blanc) / 2
```

Donc :

```
cible = 45
```

Cette valeur représente approximativement le bord entre le noir et le blanc.

5. La valeur mesurée

Pendant le déplacement, le capteur fournit :

```
valeur = capteur_couleur.reflection()
```

Cette valeur change continuellement.

Par exemple :

```
20
35
44
48
60
75
```

6. Calculer l'erreur

Nous allons calculer :

```
erreur = cible - valeur
```

Exemple :

```
cible = 45
valeur = 35
```

Alors :

```
erreur = 45 moins 35
erreur = 10
```

7. Deuxième exemple

Si :

```
cible = 45
valeur = 70
```

alors :

```
erreur = 45 moins 70
erreur = moins 25
```

Le signe de l'erreur indique de quel côté de la cible nous nous trouvons.

8. Interpréter l’erreur

On peut avoir :

```
erreur positive

↓

correction dans un sens
```

et :

```
erreur négative

↓

correction dans l’autre sens
```

Si :

```
erreur = 0
```

nous sommes exactement sur la valeur cible.

9. Exercice 1

Avec :

```
cible = 50
```

calculer l’erreur pour :

```
valeur = 40
```

Réponse :

```
50 moins 40 = 10
```

10. Exercice 2

Avec :

```
cible = 50

valeur = 65
```

Réponse :

```
50 moins 65 = moins 15
```

11. Exercice 3

Avec :

```
cible = 50
valeur = 50
```

Réponse :

```
0
```

Le robot est exactement sur la valeur souhaitée.

12. Transformer l'erreur en correction

Nous pouvons directement écrire :

```
correction = erreur
```

Mais cela donne parfois une correction trop faible ou trop forte.

Nous allons donc multiplier l'erreur par un coefficient.

```
correction = erreur * kp
```

`kp` signifie coefficient proportionnel.

13. Exemple avec `kp`

Prenons :

```
kp = 2
```

Si :

```
erreur = 5
```

alors :

```
correction = 5 × 2
correction = 10
```

Si :

```
erreur = 20
```

alors :

```
correction = 20 × 2
correction = 40
```

La correction est donc proportionnelle à l'erreur.

14. Pourquoi parle t-on de proportionnel ?

Parce que si l'erreur double, la correction double également.

Par exemple :

```
erreur = 10
correction = 20
```

Puis :

```
erreur = 20
correction = 40
```

Avec le même `kp`, la relation reste proportionnelle.

15. Premier programme proportionnel

```
cible = 45
kp = 2
vitesse = 150

while True:

    valeur = capteur_couleur.reflection()
    erreur = cible - valeur
    correction = erreur * kp
    robot.drive(vitesse, correction)
```

Ce programme est beaucoup plus simple que le contrôleur précédent.

Il n'a même plus besoin de :

```
if
```

ou :

```
else
```

pour décider du sens.

16. Pourquoi n'avons-nous plus besoin de `if` ?

Parce que le signe de l'erreur contrôle automatiquement le sens.

Par exemple :

```
erreur = 10
```

donne :

```
correction positive
```

Alors que :

```
erreur = moins 10
```

donne :

```
correction négative
```

Le robot tourne donc automatiquement dans le sens opposé.

17. Programme complet

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)

valeur_noir = 12
valeur_blanc = 78

cible = (valeur_noir + valeur_blanc) / 2

vitesse = 150
kp = 2

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(vitesse, correction)
```

18. Tester le signe

Selon le côté de la ligne suivi et le montage du robot, il est possible que le robot corrige dans le mauvais sens.

Dans ce cas, nous pouvons utiliser :

```
erreur = valeur - cible
```

au lieu de :

```
erreur = cible - valeur
```

Ou utiliser :

```
sens = -1
```

Puis :

```
correction = erreur * kp * sens
```

19. Version avec `sens`

```
cible = 45
vitesse = 150
kp = 2
sens = 1

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp * sens

    robot.drive(vitesse, correction)
```

Pour inverser le côté suivi :

```
sens = -1
```

20. Observer les valeurs

Pendant les premiers tests, il est très utile d'afficher :

```
print(valeur)
print(erreur)
print(correction)
```

Par exemple :

```
while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    print(valeur)
    print(erreur)
    print(correction)

    robot.drive(vitesse, correction)
```

21. Tableau d'exemple

Avec :

```
cible = 45

kp = 2
```

nous obtenons :

Valeur	Erreur	Correction
20	25	50
30	15	30
40	5	10
45	0	0
50	moins 5	moins 10
60	moins 15	moins 30
70	moins 25	moins 50

Ce tableau montre clairement le comportement proportionnel.

22. Exercice 4

Compléter pour :

```
cible = 50
```

```
kp = 3
```

Valeur	Erreur	Correction
30		
40		
50		
60		
70		

Correction :

Valeur	Erreur	Correction
30	20	60
40	10	30
50	0	0
60	moins 10	moins 30
70	moins 20	moins 60

23. Qu'est-ce que `kp` ?

`kp` détermine la force de réaction du robot.

Un petit `kp` donne :

```
des corrections douces
```

Un grand `kp` donne :

```
des corrections fortes
```

24. Tester différents `kp`

Commencer avec une vitesse faible.

Par exemple :

```
vitesse = 100
```

Puis tester :

```
kp = 0,5
```

```
kp = 1
```

```
kp = 2
```

```
kp = 3
```

```
kp = 5
```

Noter les observations.

kp	Comportement
0,5	
1	
2	
3	
5	

25. kp trop faible

Si kp est trop petit :

```
le robot tourne trop doucement
```

```
la correction arrive trop lentement
```

```
le robot peut sortir de la piste dans les courbes
```

Exemple :

```
kp = 0.2
```

26. kp trop élevé

Si kp est trop grand :

```
le robot surcorrige
```

```
il passe de gauche à droite
```

```
il oscille fortement
```

Exemple :

```
kp = 10
```

peut produire un comportement très instable.

27. Qu’est-ce qu’une oscillation ?

Une oscillation est un mouvement répétitif autour de la valeur souhaitée.

Par exemple :

```
trop à gauche
↓
correction forte
↓
trop à droite
↓
correction forte
↓
trop à gauche
```

Le robot zigzague continuellement.

28. Comparaison avec le contrôleur simple

Le contrôleur simple produit volontairement un zigzag.

Le contrôleur proportionnel essaie de réduire ce zigzag.

Mais si `kp` est trop élevé, les oscillations peuvent revenir.

29. Chercher un bon `kp`

Une méthode simple consiste à :

1. utiliser une vitesse faible
2. commencer avec un petit `kp`
3. augmenter progressivement
4. observer quand le suivi devient suffisamment réactif
5. éviter une valeur provoquant trop d’oscillations

30. Exercice pratique 1

Utiliser :

```
vitesse = 100
```

Tester plusieurs valeurs de `kp`.

Noter :

Essai	kp	Sortie de ligne	Oscillation	Observation
1	0,5			
2	1			
3	2			
4	3			
5	4			

Choisir le meilleur réglage.

31. Effet de la vitesse

Une fois un bon `kp` trouvé, augmenter progressivement la vitesse.

Par exemple :

```
100 mm/s
```

```
150 mm/s
```

```
200 mm/s
```

```
250 mm/s
```

```
300 mm/s
```

Il faudra peut-être modifier à nouveau `kp`.

32. Pourquoi ?

À vitesse élevée, le robot parcourt plus de distance avant que les corrections ne produisent leur effet.

Une valeur de `kp` qui fonctionne à :

```
100 mm/s
```

n’est donc pas forcément idéale à :

```
300 mm/s
```

33. Test vitesse et `kp`

Vitesse	`kp`	Stable ?	Temps
100			
150			
200			
250			
300			

L'objectif est de comprendre que les paramètres sont liés.

34. Une fonction de suivi proportionnel

```
def suivre_ligne_proportionnel(vitesse, kp):
    while True:
        valeur = capteur_couleur.reflection()
        erreur = cible - valeur
        correction = erreur * kp
        robot.drive(vitesse, correction)
```

Puis :

```
suivre_ligne_proportionnel(150, 2)
```

35. Ajouter le sens

```
def suivre_ligne_proportionnel(vitesse, kp, sens=1):
    while True:
        valeur = capteur_couleur.reflection()
        erreur = cible - valeur
        correction = erreur * kp * sens
        robot.drive(vitesse, correction)
```

36. Suivre pendant une distance

Nous pouvons également ajouter une distance.

```
def suivre_ligne_distance(distance, vitesse, kp, sens=1):

    robot.reset()

    while robot.distance() < distance:

        valeur = capteur_couleur.reflection()

        erreur = cible - valeur

        correction = erreur * kp * sens

        robot.drive(vitesse, correction)

    robot.stop()
```

Puis :

```
suivre_ligne_distance(
    2000,
    150,
    2
)
```

37. Exercice pratique 2

Faire suivre la ligne sur :

```
3 mètres
```

avec :

```
vitesse = 180 mm/s
```

Tu dois déterminer toi-même le meilleur `kp`.

38. Limiter la correction

Une grande erreur peut produire une correction très élevée.

Par exemple :

```
erreur = 50

kp = 5
```

donne :

```
correction = 250
```

Nous pouvons décider de limiter cette correction.

39. Fonction `limiter()`

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

Puis :

```
correction = limiter(
    erreur * kp,
    -150,
    150
)
```

La correction ne dépassera jamais :

```
plus ou moins 150
```

40. Programme avec limitation

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur


vitesse = 150
kp = 3

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    correction = limiter(
        correction,
        -150,
        150
    )

    robot.drive(vitesse, correction)
```

41. Pourquoi limiter ?

Cela évite qu'une mesure extrême provoque une rotation beaucoup trop forte.

C'est particulièrement utile si le robot perd complètement la ligne.

42. La zone proche de zéro

Lorsque l'erreur vaut :

```
1
```

avec :

```
kp = 1
```

la correction vaut seulement :

```
1
```

Il est possible que cette correction soit presque imperceptible.

Mais ce n'est pas forcément un problème.

Lorsque le robot est proche de la cible, une petite correction est justement souhaitable.

43. Vitesse constante

Dans notre programme :

```
robot.drive(vitesse, correction)
```

la vitesse reste toujours la même.

Seule la rotation change.

Mais nous pourrions aussi modifier la vitesse selon l'erreur.

44. Pourquoi ralentir dans les grandes erreurs ?

Si l'erreur est importante, cela peut signifier :

```
courbe serrée
```

```
robot loin de la ligne
```

Dans ce cas, ralentir peut aider.

Si l'erreur est faible :

```
ligne droite
```

```
robot bien placé
```

nous pouvons accélérer.

45. Première vitesse adaptative

```
erreur_absolue = abs(erreur)
```

`abs()` permet d'obtenir la valeur absolue.

Par exemple :

```
abs(20) = 20
```

```
abs(moins 20) = 20
```

46. Exemple

```
if abs(erreur) > 20:
    vitesse = 100
else:
    vitesse = 200
```

Donc :

```
grande erreur
```

```
↓
```

```
ralentir
```

et :

```
petite erreur
```

```
↓
```

```
accélérer
```

47. Programme adaptatif

```
kp = 2

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    if abs(erreur) > 20:

        vitesse = 100

    else:

        vitesse = 200

    robot.drive(vitesse, correction)
```

48. Exercice pratique 3

Créer trois vitesses.

```
abs(erreur) inférieur à 5

↓

250 mm/s
abs(erreur) inférieur à 20

↓

180 mm/s
sinon

↓

100 mm/s
```

Solution possible :

```
if abs(erreur) < 5:

    vitesse = 250

elif abs(erreur) < 20:

    vitesse = 180

else:

    vitesse = 100
```

49. Pourquoi utiliser `abs()` ?

Supposons :

```
erreur = moins 30
```

Cette erreur est importante.

Mais :

```
moins 30 > 20
```

est faux.

Avec :

```
abs(erreur)
```

nous obtenons :

```
30
```

Nous mesurons donc l'importance de l'erreur sans tenir compte du sens.

50. Suivi simple contre suivi proportionnel

Suivi simple

```
if valeur < cible:
    correction = 70
else:
    correction = -70
```

Avantages :

```
simple
facile à comprendre
```

Inconvénients :

```
zigzag
correction brutale
```

Suivi proportionnel

```
erreur = cible - valeur
```

```
correction = erreur * kp
```

Avantages :

```
plus fluide
```

```
plus précis
```

```
correction adaptée
```

Inconvénients :

```
réglage de kp nécessaire
```

```
peut encore osciller
```

51. L’erreur ne disparaît pas toujours complètement

Un contrôleur proportionnel peut conserver une petite erreur.

Cela arrive lorsque la correction nécessaire pour rester parfaitement sur la cible n’est pas suffisante.

C’est une des limites du contrôleur P.

Nous verrons plus tard comment la partie intégrale du PID peut aider à corriger une erreur persistante.

52. Le contrôleur peut encore osciller

Même avec un bon `kp`, le robot peut osciller dans certaines conditions.

Nous avons donc une autre question :

Comment savoir si l’erreur est en train d’augmenter ou de diminuer rapidement ?

Cette information sera fournie plus tard par la partie dérivée du PID.

53. Défi pratique 4 : comparaison chronométrée

Chaque étudiant effectue trois tours.

Premier contrôleur :

```
suivi simple
```

Deuxième contrôleur :

```
proportionnel
```

Noter :

Contrôleur	Temps	Sorties	Oscillations
Simple			
Proportionnel			

Comparer objectivement les résultats.

54. Défi pratique 5 : optimisation

Objectif :

```
effectuer un tour complet

sans quitter la ligne

le plus rapidement possible
```

Tu peux modifier :

```
vitesse
kp
```

et éventuellement :

```
correction_max
```

55. Journal de tests

Essai	Vitesse	`kp`	Correction max	Temps	Sortie
1					
2					
3					
4					
5					

Tu dois justifier ton réglage final.

56. Défi pratique 6 : vitesse adaptative

Le robot doit :

```
accélérer sur les lignes droites

ralentir dans les courbes
```

Utiliser uniquement :

```
l'importance de l'erreur
```

pour décider de la vitesse.

57. Défi final : suivi proportionnel complet

Contraintes :

1. calibration du noir et du blanc
2. calcul automatique de la cible
3. utilisation de `kp`
4. calcul de l'erreur
5. calcul de la correction
6. limitation de la correction
7. vitesse adaptative
8. parcours d'au moins 3 mètres
9. arrêt automatique à la fin

58. Exemple de solution complète

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


valeur_noir = 12
valeur_blanc = 78

cible = (
    valeur_noir
    + valeur_blanc
) / 2


kp = 2.5

correction_max = 150

distance_totale = 3000


def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

```
robot.reset()


while robot.distance() < distance_totale:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    correction = limiter(
        correction,
        -correction_max,
        correction_max
    )

    if abs(erreur) < 5:

        vitesse = 250

    elif abs(erreur) < 20:

        vitesse = 180

    else:

        vitesse = 100

    robot.drive(
        vitesse,
        correction
    )


robot.stop()
```

59. Questions de fin de cours

Question 1

Qu'est ce qu'une valeur cible ?

Question 2

Comment calculons nous l'erreur ?

Question 3

Que signifie une erreur positive ?

Question 4

Que signifie une erreur négative ?

Question 5

À quoi sert `kp` ?

Question 6

Que se passe t il généralement si `kp` est trop faible ?

Question 7

Que se passe t il si `kp` est trop élevé ?

Question 8

Pourquoi parle t on de contrôleur proportionnel ?

Question 9

À quoi sert `abs()` ?

Question 10

Pourquoi peut il être intéressant de ralentir lorsque l'erreur est importante ?

Correction

Réponse 1

La valeur que le système essaie de maintenir.

Réponse 2

Par exemple :

```
erreur = cible - valeur
```

Réponse 3

Le robot se trouve d'un côté de la valeur cible.

Réponse 4

Le robot se trouve de l'autre côté.

Réponse 5

À déterminer la force de la correction par rapport à l'erreur.

Réponse 6

Le robot peut réagir trop lentement et perdre la ligne.

Réponse 7

Le robot peut surcorriger et osciller fortement.

Réponse 8

Parce que la correction est proportionnelle à la valeur de l'erreur.

Réponse 9

À obtenir la taille d'une erreur sans tenir compte de son signe.

Réponse 10

Parce qu'une grande erreur indique souvent qu'une correction importante est nécessaire.

Difficulté fréquente

Le robot quitte immédiatement la ligne.

Première chose à vérifier :

```
le signe de la correction
```

Tester :

```
erreur = cible - valeur
```

puis éventuellement :

```
erreur = valeur - cible
```

Difficulté fréquente

Modifier en même temps :

```
vitesse
```

```
kp
```

```
cible
```

```
hauteur du capteur
```

Il devient ensuite impossible de savoir quelle modification a amélioré le résultat.

Il faut apprendre à modifier :

```
une variable à la fois
```

puis tester.

Méthode de réglage recommandée

1. calibrer le capteur
2. fixer une vitesse lente
3. régler `kp`
4. vérifier les courbes
5. augmenter la vitesse
6. ajuster légèrement `kp`
7. enregistrer les résultats

Cours 18 : Listes, mesures et historique du robot avec MicroPython

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre ce qu'est une liste en Python
2. créer une liste vide
3. ajouter des valeurs avec `append()`
4. accéder à une valeur enregistrée
5. parcourir une liste avec `for`
6. utiliser `len()`
7. calculer une moyenne
8. utiliser `min()` et `max()`
9. enregistrer les mesures d'un capteur
10. analyser les erreurs d'un suivi de ligne
11. conserver un petit historique des mesures
12. comprendre pourquoi les valeurs précédentes peuvent être utiles en robotique

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur de couleur EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. une piste noire sur fond blanc
8. un espace permettant de tester le suivi de ligne

1. Rappel du cours précédent

Lors du cours précédent, nous avons créé un contrôleur proportionnel.

Le principe était :

```
valeur = capteur_couleur.reflection()

erreur = cible - valeur

correction = erreur * kp

robot.drive(vitesse, correction)
```

Le robot effectue continuellement :

```
MESURER
↓
CALCULER L'ERREUR
↓
CALCULER LA CORRECTION
↓
CORRIGER
```

Mais une question importante apparaît.

Que deviennent les anciennes mesures ?

Pour l'instant :

```
mesure 1
↓
utilisée
↓
oubliée
↓
mesure 2
↓
utilisée
↓
oubliée
```

Aujourd'hui, nous allons apprendre à conserver plusieurs valeurs grâce aux listes.

2. Qu'est-ce qu'une liste ?

Une liste permet de stocker plusieurs valeurs dans une seule variable.

Par exemple :

```
mesures = [12, 18, 25, 40, 55]
```

La variable :

```
mesures
```

contient maintenant cinq valeurs.

On peut imaginer une liste comme une série de cases.

```
12 | 18 | 25 | 40 | 55
```

3. Pourquoi utiliser une liste ?

Jusqu'à présent, si nous voulions conserver cinq mesures, nous pouvions écrire :

```
mesure1 = 12
mesure2 = 18
mesure3 = 25
mesure4 = 40
mesure5 = 55
```

Cela fonctionne.

Mais si nous voulons conserver :

```
100 mesures
```

cette méthode devient très peu pratique.

Avec une liste :

```
mesures = []
```

nous pouvons ajouter autant de valeurs que nécessaire.

4. Créer une liste

Une liste est créée avec :

```
mesures = []
```

Les crochets indiquent une liste.

Ici, elle est vide.

Nous pouvons aussi créer directement :

```
mesures = [10, 20, 30]
```

5. Afficher une liste

```
mesures = [10, 20, 30]

print(mesures)
```

Le résultat sera proche de :

```
[10, 20, 30]
```

6. Ajouter une valeur avec `append()`

Pour ajouter une valeur :

```
mesures.append(40)
```

Exemple :

```
mesures = []

mesures.append(10)
mesures.append(20)
mesures.append(30)

print(mesures)
```

La liste contient maintenant :

```
[10, 20, 30]
```

7. Comprendre `append()`

Prenons :

```
mesures = []
```

Puis :

```
mesures.append(15)
```

Nous obtenons :

```
[15]
```

Puis :

```
mesures.append(25)
```

Nous obtenons :

```
[15, 25]
```

Puis :

```
mesures.append(35)
```

Nous obtenons :

```
[15, 25, 35]
```

`append()` ajoute donc une nouvelle valeur à la fin de la liste.

8. Exercice 1

Créer une liste vide appelée :

```
vitesses
```

Ajouter :

```
100
```

```
200
```

```
300
```

```
400
```

Solution :

```
vitesses = []

vitesses.append(100)
vitesses.append(200)
vitesses.append(300)
vitesses.append(400)

print(vitesses)
```

9. Accéder à une valeur

Chaque élément possède une position.

Attention :

Python commence à compter à :

```
0
```

Prenons :

```
mesures = [10, 20, 30, 40]
```

Nous avons :

```
Index      0    1    2    3

Valeur    10   20   30   40
```

10. Lire un élément

Pour obtenir la première valeur :

```
print(mesures[0])
```

Résultat :

```
10
```

Pour la troisième :

```
print(mesures[2])
```

Résultat :

```
30
```

11. Erreur fréquente

Si une liste contient :

```
mesures = [10, 20, 30]
```

ceci est incorrect :

```
mesures[3]
```

Pourquoi ?

Les positions sont :

```
0
```

```
1
```

```
2
```

Il n'existe pas de position `3`.

12. Exercice 2

Avec :

```
couleurs = [15, 32, 47, 68]
```

Quelle instruction affiche :

```
47
```

Réponse :

```
print(couleurs[2])
```

13. La dernière valeur

Nous pouvons également utiliser :

```
mesures[-1]
```

pour obtenir la dernière valeur.

Exemple :

```
mesures = [10, 20, 30, 40]
print(mesures[-1])
```

Résultat :

```
40
```

Cette possibilité sera très utile pour récupérer la mesure précédente.

14. Connaître la taille d’une liste

Nous utilisons :

```
len(mesures)
```

Exemple :

```
mesures = [10, 20, 30, 40]

print(len(mesures))
```

Résultat :

```
4
```

15. Ajouter des valeurs dans une boucle

```
mesures = []

for i in range(5):

    mesures.append(i)

print(mesures)
```

Résultat :

```
[0, 1, 2, 3, 4]
```

16. Enregistrer des mesures du capteur

Nous pouvons maintenant enregistrer plusieurs valeurs de réflexion.

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur_couleur = ColorSensor(Port.S3)

mesures = []

for i in range(10):

    valeur = capteur_couleur.reflection()

    mesures.append(valeur)

    wait(100)

print(mesures)
```

Le programme effectue dix mesures et les conserve.

17. Exercice pratique 1

Placer le capteur au dessus d'une surface blanche.

Enregistrer :

```
20 mesures
```

avec :

```
for i in range(20):
```

Puis afficher toute la liste.

18. Parcourir une liste

Nous pouvons utiliser :

```
for valeur in mesures:
```

Exemple :

```
mesures = [12, 15, 18, 20]
for valeur in mesures:
    print(valeur)
```

Le programme affiche chaque élément.

19. Comprendre la boucle

Avec :

```
for valeur in mesures:
```

Python prend successivement :

```
12
15
18
20
```

et place chaque valeur dans :

```
valeur
```

20. Calculer une somme

```
mesures = [10, 20, 30]

somme = 0

for valeur in mesures:

    somme = somme + valeur

print(somme)
```

Calcul :

```
0 + 10 = 10

10 + 20 = 30

30 + 30 = 60
```

Résultat :

```
60
```

21. Calculer une moyenne

La formule est :

```
moyenne = somme / nombre de valeurs
```

En Python :

```
moyenne = somme / len(mesures)
```

Programme complet :

```
mesures = [10, 20, 30]

somme = 0

for valeur in mesures:

    somme = somme + valeur

moyenne = somme / len(mesures)

print(moyenne)
```

Résultat :

```
20
```

22. Utiliser `sum()`

Python possède également :

```
sum(mesures)
```

Nous pouvons donc écrire :

```
moyenne = sum(mesures) / len(mesures)
```

Exemple :

```
mesures = [10, 20, 30]

moyenne = sum(mesures) / len(mesures)

print(moyenne)
```

23. Créer une fonction moyenne

```
def moyenne(liste):

    return sum(liste) / len(liste)
```

Puis :

```
mesures = [10, 20, 30]

print(moyenne(mesures))
```

24. Attention aux listes vides

Si :

```
mesures = []
```

alors :

```
len(mesures)
```

vaut :

```
0
```

Faire :

```
sum(mesures) / len(mesures)
```

provoquerait donc une division par zéro.

Nous pouvons protéger la fonction.

```
def moyenne(liste):

    if len(liste) == 0:
        return 0

    return sum(liste) / len(liste)
```

25. Valeur minimum

Python possède :

```
min(mesures)
```

Exemple :

```
mesures = [15, 32, 8, 60, 40]

print(min(mesures))
```

Résultat :

```
8
```

26. Valeur maximum

```
print(max(mesures))
```

Avec :

```
mesures = [15, 32, 8, 60, 40]
```

le résultat est :

```
60
```

27. Exercice 3

Avec :

```
mesures = [21, 19, 25, 40, 15]
```

trouver :

```
nombre de mesures

minimum

maximum

moyenne
```

Solution :

```
print(len(mesures))
print(min(mesures))
print(max(mesures))
print(sum(mesures) / len(mesures))
```

28. Application à la calibration

Nous avons précédemment utilisé une seule valeur pour le noir.

Par exemple :

```
valeur_noir = 12
```

Mais nous pouvons maintenant mesurer plusieurs fois.

```
mesures_noir = []

for i in range(10):

    valeur = capteur_couleur.reflection()

    mesures_noir.append(valeur)
```

Puis :

```
valeur_noir = sum(mesures_noir) / len(mesures_noir)
```

La calibration devient plus robuste.

29. Calibration du blanc

Même principe :

```
mesures_blanc = []

for i in range(10):

    valeur = capteur_couleur.reflection()

    mesures_blanc.append(valeur)
```

Puis :

```
valeur_blanc = sum(mesures_blanc) / len(mesures_blanc)
```

30. Calcul du seuil

```
cible = (
    valeur_noir
    + valeur_blanc
) / 2
```

Nous utilisons maintenant des moyennes plutôt qu'une seule mesure.

31. Pourquoi est-ce mieux ?

Une seule mesure peut être :

```
anormalement basse

anormalement haute

influencée par une petite variation
```

Plusieurs mesures permettent d'obtenir une valeur plus représentative.

32. Fonction de mesure moyenne

Créons :

```
def mesurer_moyenne(nombre):

    mesures = []

    for i in range(nombre):

        mesures.append(
            capteur_couleur.reflection()
        )

    return sum(mesures) / len(mesures)
```

Puis :

```
valeur_noir = mesurer_moyenne(10)
```

33. Ajouter une petite attente

Pour éviter de prendre toutes les mesures presque instantanément :

```
from pybricks.tools import wait
```

Puis :

```
def mesurer_moyenne(nombre):

    mesures = []

    for i in range(nombre):

        mesures.append(
            capteur_couleur.reflection()
        )

        wait(50)

    return sum(mesures) / len(mesures)
```

34. Exercice pratique 2

Créer une calibration en deux étapes.

Première étape :

```
Placer le capteur sur le noir
```

Mesurer dix fois.

Deuxième étape :

```
Placer le capteur sur le blanc
```

Mesurer dix fois.

Afficher :

```
moyenne noir

moyenne blanc

valeur cible
```

35. Enregistrer les erreurs du suivi de ligne

Reprenons :

```
erreur = cible - valeur
```

Nous pouvons enregistrer chaque erreur.

```
erreurs = []

while robot.distance() < 1000:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    erreurs.append(erreur)

    correction = erreur * kp

    robot.drive(vitesse, correction)

robot.stop()
```

36. Après le parcours

Nous pouvons analyser :

```
print(min(erreurs))
print(max(erreurs))
```

Nous obtenons les erreurs extrêmes observées.

37. Erreur moyenne

Attention.

Calculer simplement :

```
sum(erreurs) / len(erreurs)
```

peut être trompeur.

Pourquoi ?

Parce qu'une erreur positive et une erreur négative peuvent s'annuler.

Par exemple :

```
+20
```

```
et
```

```
-20
```

donnent une moyenne de :

```
0
```

Pourtant, le robot a fait deux erreurs importantes.

38. Utiliser la valeur absolue

Il est souvent plus utile de mesurer :

```
abs(erreur)
```

Nous pouvons enregistrer :

```
erreurs_absolues = []
```

Puis :

```
erreurs_absolues.append(
    abs(erreur)
)
```

39. Calculer l'erreur absolue moyenne

```
erreur_moyenne = (
    sum(erreurs_absolues)
    / len(erreurs_absolues)
)
```

Plus cette valeur est faible, plus le robot reste généralement proche de sa cible.

40. Exemple

Robot A :

```
erreur absolue moyenne = 5
```

Robot B :

```
erreur absolue moyenne = 18
```

Toutes choses égales par ailleurs, le robot A suit probablement plus précisément la valeur cible.

41. Comparer deux réglages de `kp`

Premier essai :

```
kp = 1
```

Enregistrer les erreurs.

Deuxième essai :

```
kp = 3
```

Enregistrer de nouveau les erreurs.

Comparer :

`kp`	Erreur absolue moyenne	Erreur max	Observation
1			
3			

42. Nous commençons à utiliser des données

Jusqu’à présent, le réglage pouvait être basé uniquement sur :

```
le robot semble meilleur
```

Maintenant, nous pouvons également mesurer :

```
erreur moyenne
```

```
erreur maximale
```

```
temps de parcours
```

```
sorties de piste
```

Cela rend l’optimisation plus objective.

43. Conserver seulement quelques mesures

Une boucle de robot peut effectuer énormément de lectures.

Si nous ajoutons continuellement :

```
mesures.append(valeur)
```

la liste peut devenir très grande.

Sur un ordinateur puissant, cela peut parfois être acceptable.

Sur une brique EV3 disposant de ressources limitées, il est préférable de ne pas conserver inutilement des milliers de valeurs.

44. Une petite fenêtre de mesures

Supposons que nous voulions conserver seulement les cinq dernières mesures.

```
mesures = []
```

À chaque nouvelle mesure :

```
mesures.append(valeur)
```

Puis :

```
if len(mesures) > 5:
    mesures.pop(0)
```

45. Comprendre `pop(0)`

Prenons :

```
[10, 20, 30, 40, 50]
```

Nous ajoutons :
`60`

La liste devient :

```
[10, 20, 30, 40, 50, 60]
```

Puis :
`mesures.pop(0)`

supprime le premier élément.

Nous obtenons :

```
[20, 30, 40, 50, 60]
```

Nous conservons donc uniquement les cinq valeurs les plus récentes.

46. Exercice 4

Créer une liste qui conserve seulement :

```
3 valeurs
```

Solution :

```
mesures = []

while True:

    valeur = capteur_couleur.reflection()

    mesures.append(valeur)

    if len(mesures) > 3:
        mesures.pop(0)

    print(mesures)
```

47. Calculer une moyenne glissante

Si nous conservons les cinq dernières mesures :

```
mesures = []
```

nous pouvons calculer :

```
moyenne = sum(mesures) / len(mesures)
```

à chaque tour de boucle.

On appelle cela une moyenne glissante.

48. Exemple

Mesures :

```
40

42

41

70

43
```

Une valeur isolée de :

```
70
```

peut être une perturbation.

La moyenne permet de réduire son influence.

49. Exemple de filtre simple

```
mesures = []

while True:

    valeur = capteur_couleur.reflection()

    mesures.append(valeur)

    if len(mesures) > 5:
        mesures.pop(0)

    valeur_filtree = (
        sum(mesures)
        / len(mesures)
    )

    print(valeur_filtree)
```

50. Utiliser la valeur filtrée pour le suivi

Nous pouvons remplacer :

```
valeur = capteur_couleur.reflection()
```

par une valeur moyenne.

```
mesures = []

while True:

    nouvelle_valeur = capteur_couleur.reflection()

    mesures.append(nouvelle_valeur)

    if len(mesures) > 5:
        mesures.pop(0)

    valeur = sum(mesures) / len(mesures)

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(vitesse, correction)
```

51. Est-ce toujours meilleur ?

Non.

Le filtrage réduit certaines variations.

Mais il ajoute aussi du retard.

Le robot utilise une moyenne contenant des mesures plus anciennes.

Si la piste change rapidement, la correction peut arriver trop tard.

Nous retrouvons donc un compromis entre :

```
stabilité
```

et :

```
rapidité de réaction
```

52. Expérience sur la taille de la fenêtre

Tester :

```
1 mesure
```

puis :

```
3 mesures
```

puis :

```
5 mesures
```

puis :

```
10 mesures
```

Noter :

Nombre de mesures	Stabilité	Réactivité
1		
3		
5		
10		

53. Accéder à l'erreur précédente

Nous allons maintenant utiliser une notion importante.

Si nous avons :

```
erreurs = [10, 7, 4]
```

la dernière erreur est :

```
erreurs[-1]
```

Donc :

```
4
```

Mais si nous avons déjà ajouté l'erreur actuelle, l'erreur précédente peut être :

```
erreurs[-2]
```

54. Méthode plus simple

Dans une boucle de contrôle, il est souvent plus simple d'utiliser une variable :

```
erreur_precedente = 0
```

Puis :

```
erreur = cible - valeur
```

Nous pouvons comparer :

```
difference = erreur - erreur_precedente
```

Puis, à la fin :

```
erreur_precedente = erreur
```

55. Exemple

Première mesure :

```
erreur précédente = 10
```

```
erreur actuelle = 5
```

Alors :

```
différence = 5 moins 10
```

```
=
```

```
moins 5
```

L'erreur diminue.

56. Deuxième exemple

```
erreur précédente = 5
```

```
erreur actuelle = 20
```

Alors :

```
différence = 15
```

L'erreur augmente rapidement.

57. Pourquoi cette information est-elle intéressante ?

Le contrôleur proportionnel sait uniquement :

```
où se trouve le robot maintenant
```

Avec l'erreur précédente, nous pouvons aussi savoir :

```
dans quelle direction l'erreur évolue
```

Par exemple :

```
erreur importante

mais en train de diminuer rapidement
```

n'est pas la même situation que :

```
erreur importante

et encore en train d'augmenter
```

58. Démonstration

```
cible = 45

erreur_precedente = 0

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    variation = (
        erreur
        - erreur_precedente
    )

    print("Erreur :")
    print(erreur)

    print("Variation :")
    print(variation)

    erreur_precedente = erreur
```

59. Attention à la première mesure

Au début :

```
erreur_precedente = 0
```

Cette valeur ne correspond pas forcément à une vraie mesure précédente.

Pour les exercices simples, cela ne pose généralement pas de problème.

Dans un contrôleur plus avancé, on peut initialiser soigneusement les variables avant de démarrer.

60. Somme des erreurs

Nous pouvons également accumuler les erreurs.

```
somme_erreurs = 0
```

Puis :

```
somme_erreurs = somme_erreurs + erreur
```

Après plusieurs passages :

```
erreur 1 = 5
```

```
erreur 2 = 4
```

```
erreur 3 = 3
```

nous obtenons :

```
somme = 12
```

61. Pourquoi additionner les erreurs ?

Si une petite erreur reste présente pendant longtemps, sa somme devient progressivement importante.

Exemple :

```
erreur = 2
```

```
pendant 10 mesures
```

La somme devient :

```
20
```

Même si l’erreur instantanée est faible, nous pouvons détecter qu’elle persiste.

62. Trois informations différentes

Nous avons maintenant découvert trois façons d'étudier l'erreur.

Erreur actuelle

```
erreur
```

Elle indique :

```
où nous sommes actuellement
```

Accumulation des erreurs

```
somme_erreurs
```

Elle indique :

```
si une erreur persiste dans le temps
```

Variation de l'erreur

```
erreur - erreur_precedente
```

Elle indique :

```
comment l'erreur évolue
```

63. Une première vue du PID

Ces trois informations correspondent aux trois parties d'un contrôleur très connu.

```
P

Proportionnel

↓

erreur actuelle
I

Intégral

↓

accumulation des erreurs
D

Dérivé

↓

variation de l'erreur
```

C'est le contrôleur :

PID

64. Nous connaissons déjà P

Lors du cours précédent :

```
correction = erreur * kp
```

C'était la partie :

```
P
```

Nous allons bientôt ajouter :

```
ki
```

pour l'intégrale.

Puis :

```
kd
```

pour la dérivée.

65. Aperçu de la formule

Un contrôleur PID peut être représenté ainsi :

```
Correction
=
P
+
I
+
D
```

En Python, une version simplifiée ressemble à :

```
p = erreur * kp
i = somme_erreurs * ki
d = variation * kd
correction = p + i + d
```

Nous étudierons cela progressivement.

66. Mini défi 1 : enregistreur de capteur

Enregistrer :

```
50 mesures
```

du capteur de couleur.

Afficher ensuite :

```
nombre de mesures
```

```
minimum
```

```
maximum
```

```
moyenne
```

67. Mini défi 2 : calibration automatique

Créer :

```
def mesurer_surface(nombre_mesures):
```

qui retourne la moyenne des mesures.

Exemple :

```
def mesurer_surface(nombre_mesures):

    mesures = []

    for i in range(nombre_mesures):

        mesures.append(
            capteur_couleur.reflection()
        )

        wait(50)

    return (
        sum(mesures)
        / len(mesures)
    )
```

68. Mini défi 3 : comparer deux kp

Faire deux parcours.

Premier :

```
kp = 1
```

Deuxième :

```
kp = 3
```

Pour chacun, calculer :

```
erreur absolue moyenne
```

Comparer les résultats.

69. Mini défi 4 : moyenne glissante

Créer un suivi proportionnel utilisant uniquement les :

```
3 dernières mesures
```

Comparer avec le suivi sans filtre.

70. Mini défi 5 : mesurer la variation

Pendant un suivi de ligne, afficher :

```
erreur actuelle
```

```
erreur précédente
```

```
variation
```

Observer les valeurs dans :

```
une ligne droite
```

et :

```
une courbe
```

71. Défi final : robot analyseur

Le robot doit suivre une ligne pendant :

```
2 mètres
```

Il doit enregistrer suffisamment d'informations pour afficher à la fin :

```
nombre de mesures
```

```
valeur minimum du capteur
```

```
valeur maximum
```

```
erreur absolue moyenne
```

```
erreur absolue maximum
```

72. Exemple de solution

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


valeur_noir = 12
valeur_blanc = 78

cible = (
    valeur_noir
    + valeur_blanc
) / 2


kp = 2

vitesse = 150

distance_totale = 2000


mesures = []

erreurs_absolues = []


robot.reset()


while robot.distance() < distance_totale:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    mesures.append(valeur)

    erreurs_absolues.append(
        abs(erreur)
    )

    robot.drive(
        vitesse,
        correction
    )
```

```
robot.stop()


print("Nombre de mesures :")
print(len(mesures))

print("Valeur minimum :")
print(min(mesures))

print("Valeur maximum :")
print(max(mesures))

print("Erreur moyenne :")

print(
    sum(erreurs_absolues)
    / len(erreurs_absolues)
)

print("Erreur maximale :")

print(
    max(erreurs_absolues)
)
```

73. Version plus adaptée à la mémoire

Dans un véritable programme autonome de longue durée, il n'est pas toujours nécessaire de conserver toutes les mesures.

Nous pouvons calculer certaines statistiques directement.

Par exemple :

```
nombre_mesures = 0

somme_erreurs = 0

erreur_max = 0
```

Puis :

```
nombre_mesures = nombre_mesures + 1

somme_erreurs = (
    somme_erreurs
    + abs(erreur)
)

if abs(erreur) > erreur_max:
    erreur_max = abs(erreur)
```

Cela utilise beaucoup moins de mémoire qu'une liste contenant des milliers de valeurs.

74. Exemple sans grande liste

```
nombre_mesures = 0
somme_erreurs = 0
erreur_max = 0

robot.reset()

while robot.distance() < 2000:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    nombre_mesures = nombre_mesures + 1

    somme_erreurs = (
        somme_erreurs
        + abs(erreur)
    )

    if abs(erreur) > erreur_max:

        erreur_max = abs(erreur)

    robot.drive(
        vitesse,
        correction
    )

robot.stop()

erreur_moyenne = (
    somme_erreurs
    / nombre_mesures
)

print(erreur_moyenne)
print(erreur_max)
```

75. Quand utiliser une liste ?

Une liste est utile lorsque nous avons réellement besoin de retrouver les valeurs individuelles.

Par exemple :

```
analyser les mesures après le parcours

comparer les anciennes valeurs

calculer une moyenne glissante

observer l'évolution
```

76. Quand ne pas utiliser une grande liste ?

Si nous voulons seulement connaître :

`la moyenne`

`le maximum`

`le nombre de mesures`

nous pouvons souvent calculer ces informations progressivement.

C'est particulièrement intéressant sur un système embarqué comme l'EV3.

77. Questions de fin de cours

Question 1

À quoi sert une liste ?

Question 2

Comment crée t-on une liste vide ?

Question 3

À quoi sert `append()` ?

Question 4

Que représente :

`mesures[0]`

Question 5

Que représente :

`mesures[-1]`

Question 6

À quoi sert `len()` ?

Question 7

Comment calculer une moyenne ?

Question 8

À quoi servent `min()` et `max()` ?

Question 9

Pourquoi une grande liste peut elle être problématique sur un robot ?

Question 10

Pourquoi la valeur absolue est elle utile pour mesurer l'importance d'une erreur ?

Question 11

Que représente l'erreur précédente ?

Question 12

Quelles sont les trois parties d'un PID ?

Correction

Réponse 1

Une liste permet de conserver plusieurs valeurs dans une même variable.

Réponse 2

```
liste = []
```

Réponse 3

À ajouter un élément à la fin de la liste.

Réponse 4

Le premier élément.

Réponse 5

Le dernier élément.

Réponse 6

À connaître le nombre d'éléments.

Réponse 7

Par exemple :

```
moyenne = sum(liste) / len(liste)
```

Réponse 8

À obtenir respectivement la plus petite et la plus grande valeur.

Réponse 9

Parce que la mémoire disponible est limitée.

Réponse 10

Parce qu'elle permet de mesurer la taille de l'erreur sans tenir compte de son sens.

Réponse 11

La valeur de l'erreur lors du cycle précédent.

Réponse 12

```
P = proportionnel
I = intégral
D = dérivé
```

Difficulté fréquente

Utiliser :

```
mesures[-1]
```

alors que la liste est vide.

Une liste vide ne possède pas de dernier élément.

Avant certaines opérations, il faut donc vérifier :

```
if len(mesures) > 0:
```

Difficulté fréquente

Conserver toutes les mesures sans raison.

Faire comprendre que :

```
plus de données
```

ne signifie pas automatiquement :

```
meilleur programme
```

Un robot doit utiliser efficacement ses ressources.

Cours 19 : Le contrôleur PID avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le principe général d'un contrôleur PID
2. distinguer les parties P, I et D
3. comprendre le rôle de `kp`
4. comprendre le rôle de `ki`
5. comprendre le rôle de `kd`
6. calculer une correction PID
7. utiliser l'erreur précédente
8. accumuler les erreurs
9. limiter l'intégrale
10. régler progressivement un PID
11. comparer P, PD et PID
12. appliquer un PID au suivi de ligne

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur de couleur EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. une piste avec ligne noire sur fond blanc
8. plusieurs lignes droites
9. plusieurs courbes
10. une piste complète pour les tests

1. Rappel du contrôleur proportionnel

Nous avions créé :

```
valeur = capteur_couleur.reflection()

erreur = cible - valeur

correction = erreur * kp

robot.drive(vitesse, correction)
```

Le principe était :

```
Petite erreur
↓
Petite correction
Grande erreur
↓
Grande correction
```

Cette partie correspond au :

P

de PID.

2. Que signifie PID ?

PID signifie :

```
P = Proportionnel

I = Intégral

D = Dérivé
```

Ces trois parties utilisent trois informations différentes sur l'erreur.

```
P

↓

Quelle est l'erreur maintenant ?
I

↓

Depuis combien de temps l'erreur persiste t elle ?
D

↓

À quelle vitesse l'erreur est elle en train de changer ?
```

3. Vision générale

Nous pouvons représenter le PID ainsi :

```
VALEUR CIBLE
↓
Comparer avec la mesure
↓
ERREUR
↓
P + I + D
↓
CORRECTION
↓
MOTEURS
```

Puis le robot mesure à nouveau.

```
MESURE
↓
ERREUR
↓
CORRECTION
↓
MOUVEMENT
↓
NOUVELLE MESURE
```

4. La partie P

Nous connaissons déjà :

```
p = erreur * kp
```

Par exemple :

```
erreur = 10
```

```
kp = 2
```

donne :

```
P = 20
```

5. Que fait P ?

La partie proportionnelle regarde uniquement l'erreur actuelle.

Elle ne sait pas :

```
ce qui s'est passé avant
```

Elle ne sait pas non plus :

```
si l'erreur augmente ou diminue
```

Elle répond simplement à la question :

À quelle distance suis je actuellement de ma cible ?

6. Limite de P

Prenons un robot qui oscille.

```
erreur = 20
↓
correction importante
↓
robot dépasse la cible
↓
erreur = moins 20
↓
correction importante dans l'autre sens
```

Le robot peut continuer à osciller.

Nous avons donc besoin d'informations supplémentaires.

7. La partie D

Commençons par la dérivée.

La partie D observe comment l'erreur évolue.

Nous pouvons calculer :

```
variation = erreur - erreur_precedente
```

8. Exemple

Supposons :

```
erreur précédente = 20

erreur actuelle = 10
```

Alors :

```
variation = 10 moins 20

variation = moins 10
```

Cela signifie que l'erreur diminue.

9. Deuxième exemple

```
erreur précédente = 10
erreur actuelle = 25
```

Alors :

```
variation = 15
```

L'erreur augmente rapidement.

10. Pourquoi cette information est-elle utile ?

Imaginons deux robots.

Robot A

```
erreur = 20
```

```
mais l'erreur diminue rapidement
```

Robot B

```
erreur = 20
```

```
mais l'erreur augmente rapidement
```

La valeur actuelle est identique.

Pourtant, les situations sont différentes.

La partie D permet de détecter cette différence.

11. Calcul de D

Nous utilisons :

```
d = (
    erreur
    - erreur_precedente
) * kd
```

`kd` détermine l'importance de la partie dérivée.

12. Exemple

Avec :

```
erreur précédente = 20
```

```
erreur = 10
```

```
kd = 2
```

nous avons :

```
variation = moins 10
```

Puis :

```
D = moins 10 × 2
```

```
D = moins 20
```

13. Effet général de D

La partie D agit souvent comme un amortisseur.

Elle peut aider à réduire :

```
les oscillations

les surcorrections

les changements de direction trop brutaux
```

On peut comparer cela aux amortisseurs d'une voiture.

Sans amortissement, un système peut continuer à osciller.

14. Contrôleur PD

Avant de parler de I, nous pouvons combiner :

```
P + D
```

Programme :

```
erreur_precedente = 0

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    p = erreur * kp

    d = (
        erreur
        - erreur_precedente
    ) * kd

    correction = p + d

    robot.drive(vitesse, correction)

    erreur_precedente = erreur
```

15. Ordre important

À la fin de chaque cycle :

```
erreur_precedente = erreur
```

prépare la prochaine boucle.

On peut représenter :

```
Cycle 1

erreur actuelle
↓
devient erreur précédente
↓
Cycle 2

nouvelle erreur actuelle
```

16. Exercice 1

Avec :

```
erreur = 10

erreur précédente = 6

kp = 2

kd = 3
```

Calculer P.

```
P = 10 × 2

P = 20
```

Calculer D.

```
variation = 10 moins 6

variation = 4
```

Donc :

```
D = 4 × 3

D = 12
```

Correction :

```
20 + 12 = 32
```

17. Exercice 2

Avec :

```
erreur = 10

erreur précédente = 15

kp = 2

kd = 3
```

P :

```
20
```

D :

```
10 moins 15 = moins 5

moins 5 × 3 = moins 15
```

Correction :

```
20 moins 15 = 5
```

La partie D réduit donc ici la correction car l'erreur est déjà en train de diminuer.

18. Tester P contre PD

Premier essai :

```
kp = 2
kd = 0
```

Deuxième essai :

```
kp = 2
kd = 1
```

Troisième essai :

```
kp = 2
kd = 3
```

Observer :

kp	kd	Oscillations	Réactivité
2	0		
2	1		
2	3		

19. `kd` trop faible

Si `kd` est trop faible :

```
peu d'effet

oscillations encore présentes
```

20. `kd` trop élevé

Si `kd` est trop élevé :

```
réaction très sensible aux variations

mouvements brusques

correction instable
```

Une petite variation de capteur peut produire une forte correction.

21. Les capteurs contiennent du bruit

Supposons que le capteur lise :

```
44

45

43

46

44
```

Même si le robot est presque stable.

La partie D observe les différences entre deux mesures.

Elle peut donc amplifier les petites variations.

C'est une raison pour laquelle `kd` doit être réglé avec soin.

22. La partie I

Passons maintenant à :

I

La partie intégrale accumule les erreurs.

Nous créons :

```
somme_erreurs = 0
```

Puis dans la boucle :

```
somme_erreurs = (
    somme_erreurs
    + erreur
)
```

23. Exemple d’accumulation

Supposons que l’erreur reste :

```
2
```

pendant plusieurs cycles.

Nous obtenons :

```
Cycle 1

somme = 2
Cycle 2

somme = 4
Cycle 3

somme = 6
Cycle 4

somme = 8
```

L’erreur est petite, mais elle persiste.

24. Pourquoi I est-elle utile ?

Un contrôleur proportionnel peut parfois rester légèrement décalé.

Par exemple :

```
valeur cible = 45

robot reste souvent autour de 48
```

Erreur :

```
moins 3
```

La correction P existe mais peut ne pas suffire à supprimer complètement ce décalage.

La partie intégrale accumule cette petite erreur.

Elle peut donc progressivement augmenter la correction.

25. Calcul de I

```
i = somme_erreurs * ki
```

`ki` contrôle l’importance de l’intégrale.

26. Exemple

Si :

```
somme_erreurs = 20
```

```
ki = 0.1
```

Alors :

```
I = 2
```

Si l'erreur persiste et que :

```
somme_erreurs = 100
```

alors :

```
I = 10
```

La correction intégrale augmente progressivement.

27. Contrôleur PI

Nous pouvons créer :

```
somme_erreurs = 0

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    somme_erreurs = (
        somme_erreurs
        + erreur
    )

    p = erreur * kp

    i = somme_erreurs * ki

    correction = p + i

    robot.drive(vitesse, correction)
```

28. Problème de l’intégrale

Imaginons que le robot perde la ligne pendant longtemps.

L’erreur peut rester importante.

Par exemple :

```
erreur = 30
```

pendant de nombreuses boucles.

La somme peut devenir :

```
300

600

1000

2000
```

La correction intégrale devient alors énorme.

29. Integral windup

Ce phénomène est souvent appelé :

```
integral windup
```

ou saturation de l’intégrale.

L’intégrale accumule tellement d’erreur que le système met beaucoup de temps à revenir à un comportement normal.

30. Limiter l’intégrale

Nous pouvons imposer :

```
minimum = moins 100

maximum = 100
```

Par exemple :

```
if somme_erreurs > 100:
    somme_erreurs = 100

if somme_erreurs < -100:
    somme_erreurs = -100
```

31. Utiliser notre fonction `limiter()`

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

Puis :

```
somme_erreurs = limiter(
    somme_erreurs,
    -100,
    100
)
```

32. Une autre stratégie

Nous pouvons remettre l'intégrale à zéro dans certaines situations.

Par exemple si l'erreur devient très grande :

```
if abs(erreur) > 30:
    somme_erreurs = 0
```

Cela peut éviter qu'une grosse perte de ligne provoque une intégrale excessive.

33. Enfin le PID complet

Nous avons maintenant :

```
P

I

D
```

La correction totale devient :

```
correction = p + i + d
```

34. Formule complète

```
p = erreur * kp
i = somme_erreurs * ki

d = (
    erreur
    - erreur_precedente
) * kd

correction = p + i + d
```

35. Programme PID simple

```
kp = 2
ki = 0.01
kd = 1

somme_erreurs = 0
erreur_precedente = 0

while True:

    valeur = capteur_couleur.reflection()
    erreur = cible - valeur
    somme_erreurs = (
        somme_erreurs
        + erreur
    )

    p = erreur * kp

    i = somme_erreurs * ki

    d = (
        erreur
        - erreur_precedente
    ) * kd

    correction = p + i + d

    robot.drive(
        vitesse,
        correction
    )

    erreur_precedente = erreur
```

36. Comprendre chaque ligne

```
erreur = cible - valeur
```

indique la position actuelle par rapport à la cible.

```
somme_erreurs = somme_erreurs + erreur
```

enregistre l'erreur accumulée.

```
p = erreur * kp
```

corrige selon l'erreur actuelle.

```
i = somme_erreurs * ki
```

corrige une erreur persistante.

```
d = (
    erreur
    - erreur_precedente
) * kd
```

réagit à la variation de l'erreur.

37. Mise à jour de l’erreur précédente

Il ne faut pas oublier :

```
erreur_precedente = erreur
```

Cette instruction doit généralement être placée après le calcul de D.

Sinon, la variation ne sera pas calculée correctement.

38. Exemple numérique complet

Supposons :

```
erreur = 10
```

```
erreur précédente = 6
```

```
somme des erreurs = 30
```

```
kp = 2
```

```
ki = 0,1
```

```
kd = 3
```

P

```
10 × 2 = 20
```

I

```
30 × 0,1 = 3
```

D

```
(10 moins 6) × 3
```

```
=
```

```
12
```

Correction totale

```
20 + 3 + 12
```

```
=
```

```
35
```

39. Exercice pratique 1

Avec :

```
erreur = 5

erreur précédente = 8

somme = 40

kp = 2

ki = 0,1

kd = 2
```

Calculer P.

Réponse :

```
10
```

Calculer I.

Réponse :

```
4
```

Calculer D.

```
(5 moins 8) × 2

=

moins 6
```

Correction :

```
10 + 4 moins 6

=

8
```

40. Comparaison des contrôleurs

P

```
Utilise l’erreur actuelle
```

PI

```
Utilise l’erreur actuelle

+

l’accumulation
```

PD

```
Utilise l’erreur actuelle

+

la variation
```

PID

```
Utilise les trois
```

41. Faut-il toujours utiliser PID ?

Non.

Un système plus complexe n’est pas automatiquement meilleur.

Un simple contrôleur P peut parfois suffire.

Pour un suivi de ligne EV3, un contrôleur PD peut déjà donner d’excellents résultats.

La partie I n’est pas toujours indispensable.

Le bon choix dépend du comportement du robot.

42. Méthode de réglage

Il est déconseillé de modifier :

```
kp

ki

kd
```

simultanément.

Sinon il devient très difficile de comprendre l’effet de chaque coefficient.

43. Étape 1 : régler P

Commencer avec :

```
ki = 0
kd = 0
```

Modifier uniquement :

```
kp
```

Chercher une valeur qui permet au robot de suivre la ligne.

Il peut encore osciller.

Ce n’est pas grave pour cette étape.

44. Étape 2 : ajouter D

Conserver `kp`.

Ajouter progressivement :

```
kd
```

Objectif :

```
réduire les oscillations
```

```
stabiliser le robot
```

45. Étape 3 : ajouter I seulement si nécessaire

Commencer avec une très petite valeur :

```
ki = 0.001
```

ou :

```
ki = 0.01
```

selon le comportement du robot.

Observer si une petite erreur persistante disparaît.

46. Pourquoi `ki` est souvent petit ?

La somme des erreurs peut devenir beaucoup plus grande que l'erreur instantanée.

Par exemple :

```
erreur actuelle = 5
```

mais :

```
somme des erreurs = 500
```

Même :

```
ki = 0,1
```

donnerait :

```
I = 50
```

ce qui peut être très important.

47. Réglage expérimental

Créer un tableau.

Essai	`kp`	`ki`	`kd`	Temps	Oscillations	Sortie
1						
2						
3						
4						
5						

Tu dois modifier une variable à la fois.

48. Limiter la correction finale

La somme de P, I et D peut devenir importante.

Nous pouvons donc écrire :

```
correction = limiter(
    p + i + d,
    -200,
    200
)
```

Cela évite des commandes de rotation excessives.

49. Programme avec sécurités

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur


kp = 2
ki = 0.01
kd = 1

somme_erreurs = 0
erreur_precedente = 0

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    somme_erreurs = (
        somme_erreurs
        + erreur
    )

    somme_erreurs = limiter(
        somme_erreurs,
        -500,
        500
    )

    p = erreur * kp

    i = somme_erreurs * ki

    d = (
        erreur
        - erreur_precedente
    ) * kd

    correction = p + i + d

    correction = limiter(
        correction,
        -200,
        200
    )

    robot.drive(
        vitesse,
        correction
    )

    erreur_precedente = erreur
```

50. Afficher P, I et D

Pendant le développement :

```
print("P")
print(p)

print("I")
print(i)

print("D")
print(d)
```

Cela permet de voir quelle partie du PID influence le plus la correction.

51. Expérience pédagogique

Fixer :

```
kp = 2

ki = 0

kd = 0
```

Observer.

Puis :

```
kp = 2

ki = 0

kd = 2
```

Observer.

Puis ajouter :

```
ki
```

Comparer les comportements.

52. Utiliser une fonction PID

Nous pouvons séparer le calcul.

```
def calcul_pid(
    erreur,
    somme_erreurs,
    erreur_precedente,
    kp,
    ki,
    kd
):

    p = erreur * kp

    i = somme_erreurs * ki

    d = (
        erreur
        - erreur_precedente
    ) * kd

    return p + i + d
```

53. Limite de cette fonction

La fonction ne modifie pas automatiquement :

```
somme_erreurs
```

ni :

```
erreur_precedente
```

Si tu es débutant, il peut être plus simple de conserver ces variables dans la boucle principale.

54. Une fonction pour le suivi complet

```
def suivre_ligne_pid(
    distance,
    vitesse,
    kp,
    ki,
    kd
):

    somme_erreurs = 0
    erreur_precedente = 0

    robot.reset()

    while robot.distance() < distance:

        valeur = (
            capteur_couleur.reflection()
        )

        erreur = cible - valeur

        somme_erreurs = (
```

```
            somme_erreurs
            + erreur
        )

        p = erreur * kp

        i = somme_erreurs * ki

        d = (
            erreur
            - erreur_precedente
        ) * kd

        correction = p + i + d

        robot.drive(
            vitesse,
            correction
        )

        erreur_precedente = erreur

    robot.stop()
```

Puis :

```
suivre_ligne_pid(
    3000,
    150,
    2,
    0.01,
    1
)
```

55. Contrôle de vitesse adaptatif

Nous pouvons encore utiliser :

```
abs(erreur)
```

pour adapter la vitesse.

Par exemple :

```
if abs(erreur) < 5:

    vitesse = 250

elif abs(erreur) < 20:

    vitesse = 180

else:

    vitesse = 100
```

Ainsi :

```
ligne droite

↓

accélération
grande correction

↓

ralentissement
```

56. Programme PID avec vitesse adaptative

```
kp = 2
ki = 0.01
kd = 1

somme_erreurs = 0
erreur_precedente = 0

while True:

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    somme_erreurs += erreur

    p = erreur * kp
    i = somme_erreurs * ki
    d = (
        erreur
        - erreur_precedente
    ) * kd

    correction = p + i + d

    if abs(erreur) < 5:

        vitesse = 250

    elif abs(erreur) < 20:

        vitesse = 180

    else:

        vitesse = 100

    robot.drive(
        vitesse,
        correction
    )

    erreur_precedente = erreur
```

57. Attention au temps

Notre formule simplifiée utilise :

```
somme_erreurs += erreur
```

et :

```
erreur - erreur_precedente
```

Dans un véritable PID, le temps entre les mesures est également important.

La partie intégrale dépend normalement du temps :

```
erreur × durée
```

La partie dérivée dépend de :

```
variation de l'erreur / durée
```

Pour notre robot pédagogique, nous commençons par une version simplifiée.

58. Pourquoi parler du temps ?

Imagine deux programmes.

Programme A exécute la boucle :

```
100 fois par seconde
```

Programme B :

```
10 fois par seconde
```

La somme des erreurs ne grandira pas au même rythme.

La variation calculée ne représentera pas exactement la même chose non plus.

Dans des systèmes industriels, le temps est donc intégré précisément au calcul PID.

59. Version avec période approximative

Nous pourrions choisir une petite période fixe.

Par exemple :

```
from pybricks.tools import wait
```

Puis :

```
wait(10)
```

à chaque boucle.

Cela rend la période plus régulière.

Mais cela ralentit aussi légèrement le contrôleur.

Pour nos premiers tests, nous pouvons garder le programme simple.

60. Perdre complètement la ligne

Un PID n'est pas magique.

Si le robot quitte complètement la piste et que le capteur ne voit plus que du blanc, il peut ne pas savoir où se trouve la ligne.

Nous aurons besoin plus tard de comportements supplémentaires.

Par exemple :

```
détecter la perte

↓

arrêter le PID

↓

chercher la ligne

↓

retrouver le bord

↓

redémarrer le PID
```

Cela nous amène vers les machines à états.

61. Mini défi 1 : P

Faire un tour avec uniquement :

```
P
```

Mesurer :

```
temps

nombre d'oscillations importantes

sorties de piste
```

62. Mini défi 2 : PD

Ajouter D.

Comparer les résultats.

L’objectif est de vérifier si D :

```
réduit les oscillations
```

63. Mini défi 3 : PID

Ajouter une très petite valeur de I.

Observer si le comportement s’améliore réellement.

Si ce n’est pas le cas, expliquer pourquoi il est possible de rester avec un contrôleur PD.

64. Mini défi 4 : intégrale limitée

Créer :

```
limite_integrale = 500
```

Puis empêcher :

```
somme_erreurs
```

de dépasser cette valeur.

65. Mini défi 5 : correction limitée

Créer :

```
correction_max = 200
```

et utiliser :

```
correction = limiter(
    correction,
    -correction_max,
    correction_max
)
```

66. Défi final : championnat PID

Chaque étudiant doit trouver le meilleur réglage possible pour une piste identique.

Le robot doit :

```
effectuer 3 tours
```

```
sans quitter la piste
```

```
le plus rapidement possible
```

Tu peux modifier :

```
kp
```

```
ki

kd

vitesse
```

Ils doivent noter tous leurs essais.

67. Règles du défi

Un essai est valide si :

```
le robot effectue les trois tours

sans intervention humaine
```

Des pénalités peuvent être ajoutées pour :

```
sortie complète de piste

robot touché manuellement

programme relancé
```

68. Tableau de tests

Essai	`kp`	`ki`	`kd`	Vitesse	Temps	Résultat
1						
2						
3						
4						
5						
6						

69. Exemple complet du contrôleur

```
from pybricks.ev3devices import Motor, ColorSensor
from pybricks.parameters import Port
from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

```
valeur_noir = 12
valeur_blanc = 78

cible = (
    valeur_noir
    + valeur_blanc
) / 2


kp = 2
ki = 0.01
kd = 1

vitesse = 150

limite_integrale = 500
correction_max = 200

somme_erreurs = 0
erreur_precedente = 0


def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur


robot.reset()


while robot.distance() < 3000:

    valeur = (
        capteur_couleur.reflection()
    )

    erreur = cible - valeur


    somme_erreurs += erreur

    somme_erreurs = limiter(
        somme_erreurs,
        -limite_integrale,
        limite_integrale
    )


    p = erreur * kp

    i = somme_erreurs * ki

    d = (
        erreur
        - erreur_precedente
    ) * kd


    correction = p + i + d

    correction = limiter(
```

```
            correction,
            -correction_max,
            correction_max
        )


        robot.drive(
            vitesse,
            correction
        )


        erreur_precedente = erreur


robot.stop()
```

70. Tableau récapitulatif

Partie	Utilise	Effet principal
P	Erreur actuelle	Corrige selon la distance à la cible
I	Somme des erreurs	Corrige les erreurs persistantes
D	Variation de l'erreur	Réduit souvent les oscillations

71. Tableau des coefficients

Coefficient	Si trop faible	Si trop élevé
`kp`	Réaction lente	Oscillations
`ki`	Erreur persistante	Accumulation excessive
`kd`	Peu d'amortissement	Sensibilité excessive aux variations

72. Questions de fin de cours

Question 1

Que signifie PID ?

Question 2

Quelle information utilise P ?

Question 3

Quelle information utilise I ?

Question 4

Quelle information utilise D ?

Question 5

Comment calcule t on P ?

Question 6

Comment calcule t on I ?

Question 7

Comment calcule t on D dans notre version simplifiée ?

Question 8

Pourquoi D peut elle réduire les oscillations ?

Question 9

Pourquoi faut il limiter l'intégrale ?

Question 10

Pourquoi faut il modifier un coefficient à la fois pendant le réglage ?

Question 11

Un PID est il toujours meilleur qu'un PD ?

Question 12

Pourquoi le temps entre les mesures est il important dans un véritable PID ?

Correction

Réponse 1

```
Proportionnel

Intégral

Dérivé
```

Réponse 2

L'erreur actuelle.

Réponse 3

L'accumulation des erreurs.

Réponse 4

La variation de l'erreur.

Réponse 5

```
p = erreur * kp
```

Réponse 6

```
i = somme_erreurs * ki
```

Réponse 7

```
d = (
    erreur
    - erreur_precedente
) * kd
```

Réponse 8

Parce qu'elle tient compte de la vitesse de variation de l'erreur et peut limiter les surcorrections.

Réponse 9

Pour éviter une accumulation énorme pouvant rendre le système instable.

Réponse 10

Pour pouvoir identifier clairement l'effet de chaque modification.

Réponse 11

Non.

Un contrôleur plus simple peut être suffisant ou même plus facile à régler.

Réponse 12

Parce que l'intégration et la dérivation dépendent normalement de la durée entre les mesures.

Difficulté fréquente

Tu augmentes tous les coefficients lorsque le robot fonctionne mal.

Il faut se rappeler :

```
modifier une chose

tester

observer

noter

puis modifier la suivante
```

Difficulté fréquente

Oublier :

```
erreur_precedente = erreur
```

Dans ce cas, la partie D ne représente plus correctement l'évolution d'un cycle à l'autre.

Difficulté fréquente

Mettre un `ki` trop élevé.

La partie intégrale devient rapidement dominante.

Pour un suivi de ligne, il est souvent préférable de commencer avec :

```
ki = 0
```

puis de n'ajouter I que si elle apporte réellement une amélioration.

Cours 20 : Les machines à états avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre ce qu'est un état
2. comprendre ce qu'est une machine à états
3. représenter le comportement d'un robot avec plusieurs états
4. utiliser une variable `etat`
5. changer d'état avec des conditions
6. éviter une accumulation de `if` difficile à lire
7. combiner plusieurs capteurs dans un même programme
8. créer un robot qui change de comportement selon la situation
9. comprendre la notion de transition
10. créer une architecture simple pour un robot autonome complexe

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur de couleur EV3
4. 1 capteur ultrason EV3
5. éventuellement 1 capteur tactile
6. 2 roues identiques
7. 1 ordinateur avec Visual Studio Code
8. MicroPython installé sur l'EV3
9. une piste noire sur fond blanc
10. plusieurs obstacles

1. Introduction

Nous savons maintenant créer plusieurs comportements.

Par exemple :

```
suivre une ligne
détecter un obstacle
reculer
tourner
s'arrêter
```

Mais un véritable robot autonome doit souvent choisir entre plusieurs comportements.

Par exemple :

```
Suivre la ligne
↓
Obstacle détecté
↓
Éviter l'obstacle
↓
Retrouver la ligne
↓
Recommencer à suivre
```

Si nous mettons toutes ces décisions directement dans une immense boucle, le programme peut devenir difficile à comprendre.

Nous allons donc utiliser une machine à états.

2. Qu'est-ce qu'un état ?

Un état représente la situation actuelle du robot.

Par exemple :

```
SUIVRE
```

signifie :

```
Le robot suit actuellement la ligne.
```

Un autre état pourrait être :

```
EVITER
```

signifiant :

```
Le robot est actuellement en train d'éviter un obstacle.
```

3. Exemple simple

Un robot peut avoir trois états :

```
AVANCER
RECULER
STOP
```

Une variable peut stocker l'état actuel.

```
etat = "AVANCER"
```

4. Tester l’état

Nous pouvons ensuite écrire :

```
if etat == "AVANCER":
    robot.drive(200, 0)
```

Ou :

```
elif etat == "RECULER":
    robot.drive(-200, 0)
```

Ou :

```
elif etat == "STOP":
    robot.stop()
```

5. Premier exemple de machine à états

```
etat = "AVANCER"
while True:
    if etat == "AVANCER":
        robot.drive(200, 0)
    elif etat == "RECULER":
        robot.drive(-200, 0)
    elif etat == "STOP":
        robot.stop()
```

Le robot exécute le comportement correspondant à son état.

6. Mais comment changer d’état ?

C’est ici que les capteurs interviennent.

Par exemple :

```
if capteur_ultrason.distance() < 200:
    etat = "RECULER"
```

Le robot passe donc de :

```
AVANCER
```

à :

```
RECULER
```

7. Une transition

Le passage d'un état à un autre s'appelle une transition.

Par exemple :

```
AVANCER
↓
Obstacle détecté
↓
RECULER
```

Nous pouvons représenter cela ainsi :

```
[AVANCER]
    |
    | obstacle < 200 mm
    v
[RECULER]
```

8. Pourquoi utiliser une machine à états ?

Parce qu'elle permet de séparer clairement :

```
la situation actuelle
```

de :

```
la condition permettant de changer de situation
```

Le programme devient plus facile à :

```
lire

modifier

tester

déboguer
```

9. Premier robot à deux états

Nous allons créer :

```
AVANCER

STOP
```

Le robot avance jusqu'à détecter un obstacle.

Adapter le robot construit au **cours 8** pour avoir plusieurs capteurs placés judicieusement.

Puis il s’arrête définitivement.

```
etat = "AVANCER"

while True:

    if etat == "AVANCER":

        robot.drive(200, 0)

        if capteur_ultrason.distance() < 200:

            etat = "STOP"

    elif etat == "STOP":

        robot.stop()
```

10. Comprendre l’exécution

Au début :

```
etat = AVANCER
```

Le robot entre donc ici :

```
if etat == "AVANCER":
```

Il avance.

Puis il vérifie la distance.

Si :

```
distance < 200 mm
```

alors :

```
etat = "STOP"
```

Lors du prochain passage dans la boucle :

```
elif etat == "STOP":
```

sera exécuté.

11. Exercice pratique 1

Créer deux états :

```
AVANCER
```

et :

```
RECULER
```

Le robot avance.

Si un obstacle est à moins de 20 cm, il passe en marche arrière.

Une solution possible :

```
etat = "AVANCER"
while True:
    if etat == "AVANCER":
        robot.drive(200, 0)
        if capteur_ultrason.distance() < 200:
            etat = "RECULER"
    elif etat == "RECULER":
        robot.drive(-150, 0)
```

Le problème est maintenant que le robot recule indéfiniment.

Nous devons donc prévoir une nouvelle transition.

12. Ajouter une transition de retour

Nous pouvons décider :

```
RECULER
↓
distance redevenue supérieure à 400 mm
↓
AVANCER
```

Programme :

```
etat = "AVANCER"

while True:
    distance = capteur_ultrason.distance()

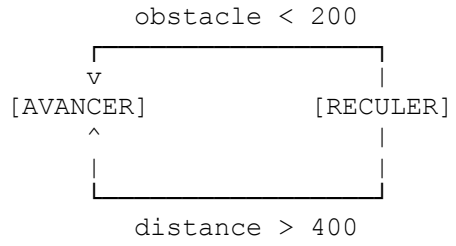
    if etat == "AVANCER":
        robot.drive(200, 0)

        if distance < 200:
            etat = "RECULER"

    elif etat == "RECULER":
        robot.drive(-150, 0)

        if distance > 400:
            etat = "AVANCER"
```

13. Schéma de cette machine



14. Pourquoi deux seuils différents ?

Nous utilisons :

```
200 mm
```

pour commencer à reculer.

Puis :

```
400 mm
```

pour recommencer à avancer.

Pourquoi ne pas utiliser 200 mm dans les deux cas ?

Parce que le robot pourrait rapidement alterner :

```
AVANCER

RECULER

AVANCER

RECULER
```

autour de la même valeur.

Utiliser deux seuils crée une zone de sécurité.

15. Notion d’hystérésis

Cette technique est appelée hystérésis.

Nous avons :

```
entrée dans RECULER à 200 mm
```

mais :

```
retour vers AVANCER à 400 mm
```

Cela évite des changements d’état trop rapides.

16. Ajouter un troisième état

Créons maintenant :

```
AVANCER
RECULER
TOURNER
```

Le comportement devient :

```
Avancer
↓
Obstacle
↓
Reculer
↓
Tourner
↓
Avancer
```

17. Comment savoir quand le recul est terminé ?

Nous pouvons utiliser `DriveBase`.

Lorsque nous entrons dans l’état `RECULER`, nous pouvons remettre la distance à zéro.

```
robot.reset()
```

Puis surveiller :

```
robot.distance()
```

18. Problème important

Nous ne devons pas écrire :

```
robot.straight(-200)
```

dans une boucle `while` exécutée continuellement.

Sinon le robot recommencerait le mouvement à chaque cycle.

Pour une machine à états, il est souvent plus pratique d'utiliser des actions continues comme :

```
robot.drive()
```

et de surveiller une condition.

19. État RECULER avec distance

Lorsque nous détectons l'obstacle :

```
robot.reset()
etat = "RECULER"
```

Puis :

```
elif etat == "RECULER":
    robot.drive(-150, 0)
    if robot.distance() <= -150:
        robot.stop()
        etat = "TOURNER"
```

Le robot recule d'environ 150 mm.

20. État TOURNER

Pour la rotation, nous pouvons utiliser l'angle accumulé par `DriveBase`.

Au moment de passer à l'état :

```
robot.reset()
etat = "TOURNER"
```

Puis :

```
elif etat == "TOURNER":
    robot.drive(0, 90)
    if robot.angle() >= 90:
        robot.stop()
        etat = "AVANCER"
```

21. Machine complète à trois états

```
etat = "AVANCER"

while True:

    if etat == "AVANCER":

        robot.drive(200, 0)

        if capteur_ultrason.distance() < 200:

            robot.stop()
            robot.reset()

            etat = "RECULER"

    elif etat == "RECULER":

        robot.drive(-150, 0)

        if robot.distance() <= -150:

            robot.stop()
            robot.reset()

            etat = "TOURNER"

    elif etat == "TOURNER":

        robot.drive(0, 90)

        if robot.angle() >= 90:

            robot.stop()

            etat = "AVANCER"
```

22. Lire le programme comme une histoire

Ce programme peut être lu ainsi :

```
État AVANCER

Le robot avance.

S'il détecte un obstacle :

passer à RECULER.
État RECULER

Le robot recule.

Lorsqu'il a suffisamment reculé :

passer à TOURNER.
État TOURNER

Le robot tourne.

Lorsqu'il a tourné de 90° :

passer à AVANCER.
```

23. Exercice pratique 2

Dessiner la machine à états du programme précédent.

Une représentation possible :

```
[AVANCER]
    |
    | obstacle
    v
[RECULER]
    |
    | 150 mm
    v
[TOURNER]
    |
    | 90°
    v
[AVANCER]
```

24. Ajouter le suivi de ligne

Nous allons maintenant créer un robot avec :

```
SUIVRE
```

et :

```
EVITER
```

Lorsque le robot suit la ligne, il utilise notre contrôleur proportionnel.

```
valeur = capteur_couleur.reflection()

erreur = cible - valeur

correction = erreur * kp

robot.drive(vitesse, correction)
```

25. État SUIVRE

```
if etat == "SUIVRE":

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(
        vitesse,
        correction
    )
```

26. Transition vers EVITER

Pendant le suivi, nous surveillons aussi l'ultrason.

```
if capteur_ultrason.distance() < 200:

    robot.stop()

    etat = "EVITER"
```

27. Mais l'évitement possède plusieurs étapes

Pour contourner un obstacle, nous devons :

```
reculer

tourner

avancer

tourner à nouveau

chercher la ligne
```

Créer un seul état `EVITER` pourrait devenir compliqué.

Il est souvent préférable de créer plusieurs états.

28. Exemple d'états

```
SUIVRE
RECULER
TOURNER_1
CONTOURNER
CHERCHER_LIGNE
```

Puis revenir à :

```
SUIVRE
```

29. Schéma général

```
[SUIVRE]
    |
    | obstacle
    v
[RECULER]
    |
    v
[TOURNER_1]
    |
    v
[CONTOURNER]
    |
    v
[CHERCHER_LIGNE]
    |
    | ligne retrouvée
    v
[SUIVRE]
```

30. Une machine à états permet de découper le problème

Au lieu de réfléchir à :

Comment contourner entièrement un obstacle ?

Nous pouvons réfléchir séparément :

```
Comment reculer ?
```

Puis :

```
Comment tourner ?
```

Puis :

```
Comment chercher la ligne ?
```

Chaque problème devient beaucoup plus simple.

31. Créer des constantes d’état

Écrire directement :

```
etat = "SUIVRE"
```

fonctionne.

Mais nous pouvons réduire les erreurs de frappe avec :

```
SUIVRE = 0
RECULER = 1
TOURNER = 2
CHERCHER = 3
STOP = 4
```

Puis :

```
etat = SUIVRE
```

32. Utiliser les constantes

```
if etat == SUIVRE:

    # suivi de ligne

elif etat == RECULER:

    # recul

elif etat == TOURNER:

    # rotation
```

Cela évite par exemple d’écrire accidentellement :

```
"SUIVREE"
```

à un endroit du programme.

33. Quelle méthode choisir ?

Pour apprendre, utiliser des mots comme :

```
etat = "SUIVRE"
```

est très lisible.

Dans un programme plus structuré, des constantes numériques peuvent être pratiques.

Les deux méthodes sont valides pour nos exercices.

34. Afficher les changements d’état

Pendant le débogage, nous pouvons afficher :

```
print("SUIVRE")
```

ou :

```
print("RECULER")
```

chaque fois que l’état change.

Cela permet de voir ce que le robot pense faire.

35. Exemple

```
if distance < 200:
    print("Obstacle détecté")
    print("Passage vers RECULER")
    robot.reset()
    etat = "RECULER"
```

Si le robot possède un comportement incorrect, nous pouvons vérifier si le problème vient :

```
de la transition
```

ou :

```
de l’action réalisée dans l’état
```

36. Fonction de changement d’état

Nous pouvons même créer :

```
def changer_etat(nouvel_etat):
    print(nouvel_etat)
    return nouvel_etat
```

Puis :

```
etat = changer_etat("RECULER")
```

Pour notre cours, cette fonction est optionnelle.

37. Machine avec capteur tactile

Le capteur tactile peut également créer une transition.

Par exemple :

```
AVANCER

↓

bouton appuyé

↓

RECULER
```

Code :

```
if capteur_tactile.pressed():

    etat = "RECULER"
```

La logique des machines à états ne dépend donc pas du type de capteur.

38. Chaque capteur peut déclencher une transition

Exemples :

```
ultrason

↓

obstacle proche

↓

EVITER
couleur rouge

↓

STOP
ligne retrouvée

↓

SUIVRE
bouton appuyé

↓

RECULER
```

Les capteurs deviennent des déclencheurs.

39. Exemple avec priorité au rouge

Supposons que le robot doit toujours s'arrêter sur une zone rouge.

Nous pouvons vérifier au début de chaque boucle :

```
if capteur_couleur.color() == Color.RED:
    etat = "STOP"
```

Puis la machine traite l'état.

40. État STOP

```
elif etat == "STOP":
    robot.stop()
```

Mais comme la boucle continue, cet état est vérifié continuellement.

Nous pouvons aussi sortir complètement du programme avec :

```
break
```

41. STOP avec `break`

```
if etat == "STOP":
    robot.stop()
    break
```

La boucle principale se termine.

42. Un robot autonome plus complet

Imaginons :

```
SUIVRE
↓
Obstacle
↓
EVITER
↓
Retrouver ligne
↓
SUIVRE
↓
Rouge
↓
STOP
```

Nous avons maintenant un véritable comportement multi étapes.

43. Fonction de suivi proportionnel pour un cycle

Attention.

Nous ne devons pas créer :

```
def suivre_ligne():

    while True:
```

dans notre machine à états.

Pourquoi ?

Parce que la fonction ne rendrait jamais la main à la machine.

Elle ne pourrait plus vérifier les autres états correctement.

44. Un cycle seulement

Dans une machine à états, il est préférable de créer une fonction qui réalise une seule correction.

```
def suivre_ligne_cycle():

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(
        vitesse,
        correction
    )
```

La boucle principale rappelle cette fonction continuellement.

45. Programme principal

```
while True:

    if etat == "SUIVRE":

        suivre_ligne_cycle()

        if capteur_ultrason.distance() < 200:

            etat = "RECULER"
```

Ainsi, chaque cycle de la boucle permet de vérifier :

```
le suivi

la distance

les transitions
```

46. Principe important

Une machine à états efficace doit éviter les fonctions qui bloquent longtemps.

Par exemple :

```
robot.straight(1000)
```

bloque le programme pendant le déplacement.

Pendant ce temps, le robot ne peut pas vérifier facilement :

```
un obstacle
```

```
une couleur
```

```
une urgence
```

Pour un robot très autonome, les actions continues avec vérification régulière sont souvent plus adaptées.

47. Comparaison

Action bloquante

```
robot.straight(1000)
```

Le robot exécute toute l'action avant de continuer.

Action non bloquante dans notre logique

```
robot.drive(200, 0)
```

Puis la boucle continue immédiatement.

Nous pouvons donc continuer à surveiller les capteurs.

48. Exercice de réflexion

Pourquoi ce programme est il mauvais pour un robot devant s'arrêter immédiatement sur un obstacle ?

```
robot.straight(5000)
if capteur_ultrason.distance() < 200:
    robot.stop()
```

Réponse :

Le capteur n'est vérifié qu'après le déplacement de 5 mètres.

49. Meilleure solution

```
robot.drive(200, 0)

while capteur_ultrason.distance() >= 200:

    pass

robot.stop()
```

Ou encore mieux dans une machine à états :

```
while True:

    if etat == "AVANCER":

        robot.drive(200, 0)

        if capteur_ultrason.distance() < 200:

            robot.stop()

            etat = "EVITER"
```

50. Ajouter un compteur d’obstacles

Nous pouvons créer :

```
compteur_obstacles = 0
```

Lors de la transition :

```
compteur_obstacles += 1
```

Puis décider :

```
if compteur_obstacles >= 5:

    etat = "STOP"
```

51. Exemple

```
if distance < 200:

    compteur_obstacles += 1

    if compteur_obstacles >= 5:

        etat = "STOP"

    else:

        robot.reset()

        etat = "RECULER"
```

La machine peut donc tenir compte de son historique.

52. Défi pratique 1

Créer une machine avec :

```
AVANCER
RECULER
TOURNER
STOP
```

Le robot doit s'arrêter définitivement après trois obstacles.

53. Une solution possible

```
etat = "AVANCER"

compteur = 0

while True:

    if etat == "AVANCER":
        robot.drive(200, 0)

        if capteur_ultrason.distance() < 200:
            robot.stop()
            compteur += 1

            if compteur >= 3:
                etat = "STOP"

            else:
                robot.reset()
                etat = "RECULER"

    elif etat == "RECULER":
        robot.drive(-150, 0)

        if robot.distance() <= -150:
            robot.stop()
            robot.reset()
            etat = "TOURNER"

    elif etat == "TOURNER":
        robot.drive(0, 90)

        if robot.angle() >= 90:
            robot.stop()
            etat = "AVANCER"

    elif etat == "STOP":
        robot.stop()
        break
```

54. Défi pratique 2

Ajouter une zone rouge.

Peu importe l'état du robot, si le rouge est détecté :

```
STOP
```

Une possibilité est de vérifier avant les états :

```
if capteur_couleur.color() == Color.RED:

    etat = "STOP"
```

55. Défi pratique 3

Créer trois états :

```
CHERCHER_LIGNE
SUIVRE_LIGNE
STOP
```

En état `CHERCHER_LIGNE`, le robot avance lentement.

Lorsqu'il trouve une surface suffisamment sombre, il passe à `SUIVRE_LIGNE`.

56. Exemple

```
if etat == "CHERCHER_LIGNE":

    robot.drive(100, 0)

    if capteur_couleur.reflection() < seuil:

        etat = "SUIVRE_LIGNE"
```

Puis :

```
elif etat == "SUIVRE_LIGNE":

    suivre_ligne_cycle()
```

57. Mais comment détecter une ligne perdue ?

Nous pouvons définir une valeur représentant une surface très claire.

Par exemple :

```
limite_perte = 75
```

Si le capteur reste très clair pendant longtemps, cela peut indiquer que le robot a quitté la ligne.

Une seule mesure ne suffit pas toujours.

58. Compteur de perte

Nous pouvons créer :

```
compteur_perte = 0
```

Puis :

```
if valeur > limite_perte:
    compteur_perte += 1
else:
    compteur_perte = 0
```

Si :

```
compteur_perte > 20
```

nous pouvons décider que la ligne est réellement perdue.

59. Transition vers CHERCHER

```
if compteur_perte > 20:
    robot.stop()
    compteur_perte = 0
    etat = "CHERCHER_LIGNE"
```

Cela évite de changer d'état à cause d'une seule mauvaise mesure.

60. Un état de recherche

Lorsque la ligne est perdue, le robot peut tourner lentement.

```
elif etat == "CHERCHER_LIGNE":
    robot.drive(0, 60)
    valeur = capteur_couleur.reflection()
    if valeur < seuil:
        robot.stop()
        etat = "SUIVRE_LIGNE"
```

61. Le robot se récupère lui même

Nous avons maintenant :

```
SUIVRE
↓
ligne perdue
↓
CHERCHER
↓
ligne détectée
↓
SUIVRE
```

Le robot possède donc une stratégie de récupération.

62. Pourquoi est-ce important ?

Un robot réel rencontre des erreurs.

Un bon programme autonome ne doit pas simplement espérer que tout fonctionne parfaitement.

Il doit aussi savoir :

```
détecter certaines erreurs

réagir

essayer de récupérer
```

63. Défi pratique 4

Créer un robot qui suit une ligne.

Si la ligne est perdue pendant plusieurs cycles :

```
CHERCHER
```

Le robot tourne jusqu'à retrouver une surface sombre.

Puis il reprend le suivi.

64. Défi pratique 5

Ajouter le capteur ultrason.

Les états deviennent :

```
SUIVRE
EVITER
CHERCHER
STOP
```

Le robot doit :

```
suivre une ligne

éviter un obstacle

retrouver la ligne

reprendre le suivi
```

65. Décomposer EVITER

Pour un meilleur programme, nous pouvons remplacer `EVITER` par :

```
RECULER

TOURNER

CONTOURNER
```

Cela donne :

```
SUIVRE
↓
RECULER
↓
TOURNER
↓
CONTOURNER
↓
CHERCHER
↓
SUIVRE
```

66. Exemple de structure générale

```
etat = "SUIVRE"

while True:

    if etat == "SUIVRE":
        # suivre la ligne
        # vérifier obstacle

    elif etat == "RECULER":
        # reculer

    elif etat == "TOURNER":
        # tourner

    elif etat == "CONTOURNER":
        # avancer autour de l'obstacle

    elif etat == "CHERCHER":
        # rechercher la ligne

    elif etat == "STOP":
        robot.stop()
        break
```

Cette structure peut contenir un programme robotique très complexe tout en restant organisée.

67. Défi final : robot autonome multi états

Créer un robot possédant au minimum cinq états :

`SUIVRE`

`RECULER`

`TOURNER`

`CHERCHER`

`STOP`

Le robot doit :

1. suivre une ligne
2. détecter un obstacle
3. quitter temporairement la ligne pour éviter l'obstacle
4. rechercher la ligne
5. reprendre le suivi
6. s'arrêter sur une zone rouge

68. Contraintes du défi

Le programme doit utiliser :

1. une variable `etat`
2. au moins cinq états
3. au moins deux capteurs
4. au moins une fonction
5. `if` et `elif`
6. au moins une boucle
7. au moins une transition basée sur une distance
8. au moins une transition basée sur une mesure de couleur ou de réflexion
9. un état `STOP`

69. Exemple de squelette

```
from pybricks.ev3devices import (
    Motor,
    ColorSensor,
    UltrasonicSensor
)

from pybricks.parameters import (
    Port,
    Color
)

from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_couleur = ColorSensor(Port.S3)
```

```
capteur_ultrason = UltrasonicSensor(Port.S4)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


valeur_noir = 12
valeur_blanc = 78

cible = (
    valeur_noir
    + valeur_blanc
) / 2


kp = 2
vitesse = 150


def suivi_cycle():

    valeur = (
        capteur_couleur.reflection()
    )

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(
        vitesse,
        correction
    )


etat = "SUIVRE"


while True:

    if capteur_couleur.color() == Color.RED:

        etat = "STOP"


    if etat == "SUIVRE":

        suivi_cycle()

        if capteur_ultrason.distance() < 200:

            robot.stop()
            robot.reset()

            etat = "RECULER"


    elif etat == "RECULER":

        robot.drive(-150, 0)

        if robot.distance() <= -150:
```

```
            robot.stop()
            robot.reset()

            etat = "TOURNER"


    elif etat == "TOURNER":

        robot.drive(0, 90)

        if robot.angle() >= 90:

            robot.stop()

            etat = "CHERCHER"


    elif etat == "CHERCHER":

        robot.drive(100, -40)

        if (
            capteur_couleur.reflection()
            < cible
        ):

            robot.stop()

            etat = "SUIVRE"


    elif etat == "STOP":

        robot.stop()

        break
```

Ce programme est volontairement simple.

L'évitement réel dépendra fortement de la forme du parcours et de la position de l'obstacle.

70. Pourquoi le squelette n'est-il pas universel ?

Parce qu'un robot réel dépend de :

```
la géométrie de la piste

la taille des obstacles

la position des capteurs

la vitesse

les dimensions du robot
```

La machine à états fournit l'architecture.

Les mouvements précis doivent ensuite être adaptés au problème.

71. Tableau de conception

Avant de programmer, tu peux compléter :

État	Action	Condition de sortie	État suivant
SUIVRE			
RECULER			
TOURNER			
CHERCHER			
STOP			

Cette étape est fortement recommandée.

72. Exemple complété

État	Action	Condition de sortie	État suivant
SUIVRE	Suivre la ligne	Obstacle sous 200 mm	RECULER
RECULER	Reculer	150 mm parcourus	TOURNER
TOURNER	Rotation	90°	CHERCHER
CHERCHER	Chercher ligne	Surface sombre détectée	SUIVRE
STOP	Arrêter	Aucune	Fin

73. Questions de fin de cours

Question 1

Qu'est ce qu'un état ?

Question 2

Qu'est ce qu'une transition ?

Question 3

À quoi sert une variable `etat` ?

Question 4

Pourquoi utiliser une machine à états ?

Question 5

Que peut déclencher une transition ?

Question 6

Pourquoi vaut il mieux éviter de longues actions bloquantes dans une machine à états ?

Question 7

Quelle différence existe entre une action et une transition ?

Question 8

Pourquoi créer un état `CHERCHER` ?

Question 9

À quoi sert un état `STOP` ?

Question 10

Qu'est ce que l'hystérésis ?

Correction

Réponse 1

Un état représente la situation ou le comportement actuel du robot.

Réponse 2

Le passage d'un état à un autre lorsqu'une condition est remplie.

Réponse 3

À mémoriser le comportement que le robot doit actuellement exécuter.

Réponse 4

Pour organiser clairement un programme possédant plusieurs comportements.

Réponse 5

Par exemple :

```
une distance

une couleur

un bouton

une durée

une distance parcourue

un angle
```

Réponse 6

Parce que le robot doit pouvoir vérifier régulièrement ses capteurs et ses conditions de transition.

Réponse 7

Une action décrit ce que fait le robot dans l'état.

Une transition décrit quand il doit quitter cet état.

Réponse 8

Pour permettre au robot de retrouver la ligne après l'avoir perdue.

Réponse 9

À définir clairement le comportement lorsque le robot doit terminer sa mission.

Réponse 10

L’utilisation de seuils différents pour entrer et sortir d’un état afin d’éviter des changements rapides et répétés.

Difficulté fréquente

Créer énormément d’états inutiles.

Un état doit représenter une véritable phase du comportement.

Il n’est pas nécessaire de créer un nouvel état pour chaque ligne de code.

Difficulté fréquente

Ne jamais quitter un état.

Par exemple :

```
elif etat == "RECULER":
    robot.drive(-150, 0)
```

sans aucune condition permettant de modifier `etat`.

Le robot restera alors dans `RECULER` indéfiniment.

Pour chaque état, demander :

Comment entre t on dans cet état ?

et :

Comment en sort on ?

Difficulté fréquente

Utiliser des fonctions contenant :

```
while True:
```

à l’intérieur d’une machine à états.

La fonction peut alors monopoliser le programme et empêcher les autres états d’être évalués.

Préférer des fonctions effectuant un seul cycle de calcul.

Cours 21 : Débogage et diagnostic d'un robot LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre ce qu'est un bug
2. distinguer plusieurs types d'erreurs
3. lire un message d'erreur Python
4. utiliser `print()` pour diagnostiquer un programme
5. tester séparément les composants du robot
6. vérifier les ports moteurs et capteurs
7. identifier une erreur de logique
8. identifier une erreur de calibration
9. identifier un problème mécanique
10. utiliser une méthode de diagnostic systématique
11. créer un programme de test du robot
12. corriger un programme complexe sans tout réécrire

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur tactile
4. 1 capteur ultrason
5. 1 capteur de couleur
6. 2 roues identiques
7. 1 ordinateur avec Visual Studio Code
8. MicroPython installé sur l'EV3
9. une piste de test
10. plusieurs obstacles

1. Introduction

Lorsque nous programmons un robot, il arrive souvent que quelque chose ne fonctionne pas comme prévu.

Par exemple :

```
le programme refuse de démarrer
```

ou :

```
le robot tourne du mauvais côté
```

ou :

```
le capteur retourne une valeur étrange
```

ou encore :

```
le robot fonctionne parfois mais pas toujours
```

Trouver la cause de ces problèmes s'appelle :

le débogage

2. Qu'est-ce qu'un bug ?

Un bug est un problème qui empêche le programme ou le robot de fonctionner comme prévu.

Mais tous les bugs ne sont pas du même type.

Nous pouvons distinguer plusieurs catégories.

```
ERREUR DE SYNTAXE

ERREUR D'EXÉCUTION

ERREUR DE LOGIQUE

ERREUR DE CONFIGURATION

ERREUR DE CALIBRATION

PROBLÈME MÉCANIQUE
```

3. Erreur de syntaxe

Une erreur de syntaxe signifie que Python ne comprend pas la manière dont le programme est écrit.

Exemple :

```
if distance < 200
    robot.stop()
```

Il manque :

```
:
```

Correction :

```
if distance < 200:
    robot.stop()
```

4. Autre erreur de syntaxe

Incorrect :

```
def avancer()
    robot.straight(500)
```

Correct :

```
def avancer():
    robot.straight(500)
```

5. Erreur d'indentation

Incorrect :

```
if distance < 200:
robot.stop()
```

Correct :

```
if distance < 200:
    robot.stop()
```

Python utilise l'indentation pour comprendre quelles instructions appartiennent à une condition, une boucle ou une fonction.

6. Exercice 1

Corriger :

```
while True

if capteur.pressed():
print("STOP")
```

Solution :

```
while True:

    if capteur.pressed():
        print("STOP")
```

7. Erreur d'exécution

Un programme peut être correctement écrit mais provoquer une erreur pendant son fonctionnement.

Exemple :

```
valeur = 10 / 0
```

Python ne peut pas diviser par zéro.

Autre exemple :

```
mesures = []

print(mesures[0])
```

La liste est vide.

Il n'existe donc aucun premier élément.

8. Mauvais nom de variable

Exemple :

```
vitesse = 200

robot.drive(vitese, 0)
```

Nous avons créé :

```
vitesse
```

mais utilisé :

```
vitese
```

Python considère qu'il s'agit de deux noms différents.

9. Sensibilité aux majuscules

Python distingue :

```
vitesse
```

de :

```
Vitesse
```

et de :

```
VITESSE
```

Ce sont trois variables différentes.

10. Erreur de logique

Une erreur de logique est plus difficile à repérer.

Le programme fonctionne sans message d'erreur, mais le résultat est incorrect.

Exemple :

```
distance_cm = 50

distance_mm = distance_cm / 10
```

Le programme fonctionne.

Mais le calcul est faux.

La conversion correcte est :

```
distance_mm = distance_cm * 10
```

11. Exemple robotique

Nous voulons tourner de 90 degrés vers la droite.

Mais nous écrivons :

```
robot.turn(-90)
```

Si le signe ne correspond pas au sens voulu, le robot tourne dans la mauvaise direction.

Le programme est valide.

C'est une erreur de logique.

12. Erreur de configuration

Supposons que le capteur ultrason soit physiquement connecté au port 4.

Mais le programme contient :

```
capteur_ultrason = UltrasonicSensor(Port.S2)
```

Le problème n'est pas forcément dans l'algorithme.

Le programme ne correspond simplement pas au branchement réel.

13. Vérifier les ports

Avant tout test, vérifier :

```
Moteur gauche = B

Moteur droit = C

Capteur couleur = S3

Capteur ultrason = S4

Capteur tactile = S1
```

Les ports utilisés dans le programme doivent correspondre au robot.

14. Erreur de calibration

Supposons :

```
valeur_noir = 10
valeur_blanc = 80
```

mais que sur la piste réelle nous avons :

```
Noir = 25

Blanc = 55
```

Notre seuil calculé à partir des anciennes valeurs peut être mauvais.

Le programme Python est correct.

Le problème vient de la calibration.

15. Problème mécanique

Parfois, le programme est parfaitement correct.

Mais :

```
une roue est mal fixée

un câble frotte sur une roue

un moteur est mal monté

le capteur bouge

une roue patine

le robot est déséquilibré
```

Dans ce cas, modifier le code ne résoudra pas forcément le problème.

16. Règle importante

Avant de modifier le programme, il faut d'abord identifier :

```
QUEL TYPE DE PROBLÈME
```

nous rencontrons.

Sinon, nous risquons de modifier du code correct pour compenser un problème mécanique.

17. Méthode de diagnostic

Lorsqu'un robot ne fonctionne pas :

```
1. Observer le problème
2. Reproduire le problème
3. Isoler la partie concernée
4. Tester cette partie seule
5. Vérifier les valeurs
6. Modifier une seule chose
7. Tester à nouveau
```

18. Étape 1 : observer

Éviter de dire seulement :

```
Ça ne marche pas.
```

Il faut décrire précisément le problème.

Par exemple :

```
Le moteur gauche ne tourne pas.
```

ou :

```
Le capteur ultrason indique toujours 2550 mm.
```

ou :

```
Le robot perd la ligne uniquement dans les virages serrés.
```

Une bonne description aide énormément au diagnostic.

19. Étape 2 : reproduire

Un problème reproductible est plus facile à analyser.

Par exemple :

```
À chaque fois que le robot arrive dans cette courbe, il quitte la ligne.
```

C’est beaucoup plus utile que :

```
Parfois ça ne fonctionne pas.
```

20. Étape 3 : isoler

Si le robot complet ne fonctionne pas, tester séparément :

```
moteur gauche

moteur droit

capteur couleur

capteur ultrason

capteur tactile
```

Ne pas tester immédiatement l’ensemble du programme.

21. Programme de test du moteur gauche

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_gauche = Motor(Port.B)

moteur_gauche.run(300)

wait(2000)

moteur_gauche.stop()
```

Question :

Le moteur tourne t il correctement ?

22. Test du moteur droit

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

moteur_droit = Motor(Port.C)

moteur_droit.run(300)

wait(2000)

moteur_droit.stop()
```

23. Tester les deux moteurs

```
moteur_gauche.run(300)
moteur_droit.run(300)

wait(2000)

moteur_gauche.stop()
moteur_droit.stop()
```

Observer :

```
les deux tournent ils ?

dans le bon sens ?

à une vitesse similaire ?
```

24. Test du capteur tactile

```
from pybricks.ev3devices import TouchSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur = TouchSensor(Port.S1)

while True:

    print(capteur.pressed())

    wait(200)
```

Appuyer et relâcher plusieurs fois.

Nous devons voir alterner :

```
True

False
```

25. Test du capteur ultrason

```
from pybricks.ev3devices import UltrasonicSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur = UltrasonicSensor(Port.S4)

while True:

    print(capteur.distance())

    wait(200)
```

Déplacer un objet devant le capteur.

La valeur doit évoluer de manière cohérente.

26. Test du capteur de couleur

```
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

capteur = ColorSensor(Port.S3)

while True:

    print(capteur.reflection())

    wait(200)
```

Tester :

```
noir

blanc
```

Les valeurs doivent être clairement différentes.

27. Pourquoi créer de petits programmes de test ?

Supposons que notre robot autonome contient :

```
400 lignes
```

Si le capteur ultrason semble incorrect, il est inutile de déboguer les 400 lignes.

Nous pouvons créer un programme de dix lignes qui teste uniquement le capteur.

Si le problème existe encore, nous savons que le problème se situe probablement dans :

```
le capteur

le branchement

le port

ou la configuration
```

28. Utiliser `print()`

`print()` est l'un des outils de débogage les plus simples.

Exemple :

```
distance = capteur_ultrason.distance()

print(distance)
```

Cela permet de vérifier ce que le programme reçoit réellement.

29. Afficher une variable calculée

Pour un suivi de ligne :

```
valeur = capteur_couleur.reflection()

erreur = cible - valeur

correction = erreur * kp

print(valeur)
print(erreur)
print(correction)
```

Nous pouvons vérifier les différentes étapes du calcul.

30. Exemple de diagnostic

Le robot tourne brutalement.

Nous observons :

```
valeur = 40

cible = 45

erreur = 5

correction = 500
```

Cela semble étrange.

Si :

```
erreur = 5
```

mais :

```
correction = 500
```

alors probablement :

```
kp = 100
```

est beaucoup trop élevé.

31. Afficher avec des textes explicites

Au lieu de :

```
print(valeur)
print(erreur)
print(correction)
```

nous pouvons utiliser :

```
print("Valeur :")
print(valeur)

print("Erreur :")
print(erreur)

print("Correction :")
print(correction)
```

Cela est plus facile à lire.

32. Déboguer une machine à états

Dans une machine à états, afficher l'état est extrêmement utile.

Par exemple :

```
print(etat)
```

Ou uniquement lors d'une transition :

```
print("SUIVRE vers RECULER")

etat = "RECULER"
```

33. Exemple

Le robot ne semble jamais éviter l'obstacle.

Nous pouvons écrire :

```
distance = capteur_ultrason.distance()

print("Distance :")
print(distance)

if distance < 200:

    print("Obstacle détecté")

    etat = "RECULER"
```

Nous pouvons alors déterminer si :

```
la distance n'atteint jamais 200
```

ou si :

```
la transition fonctionne mais RECULER possède un problème
```

34. Séparer mesure, décision et action

Une bonne méthode consiste à tester séparément :

```
MESURE
DÉCISION
ACTION
```

35. Exemple ultrason

Mesure

```
distance = capteur_ultrason.distance()

print(distance)
```

Décision

```
if distance < 200:

    print("STOP")
```

Action

```
robot.stop()
```

Tester les trois étapes permet de trouver plus rapidement la cause d'un problème.

36. Utiliser des fonctions de diagnostic

Nous pouvons créer :

```
def afficher_capteurs():

    print("Couleur :")
    print(
        capteur_couleur.reflection()
    )

    print("Distance :")
    print(
        capteur_ultrason.distance()
    )
```

Puis appeler :

```
afficher_capteurs()
```

37. Diagnostic des moteurs

Nous pouvons créer :

```
def tester_moteur(moteur):

    moteur.run(300)

    wait(1000)

    moteur.stop()
```

Puis :

```
tester_moteur(moteur_gauche)

tester_moteur(moteur_droit)
```

38. Programme de diagnostic au démarrage

Un robot complexe peut commencer par vérifier certains composants.

Par exemple :

```
print("Test moteur gauche")

tester_moteur(moteur_gauche)

print("Test moteur droit")

tester_moteur(moteur_droit)

print("Capteur couleur")

print(
    capteur_couleur.reflection()
)
```

```
print("Ultrason")

print(
    capteur_ultrason.distance()
)
```

39. Attention

Un programme de diagnostic automatique ne peut pas toujours déterminer si la valeur est correcte.

Par exemple :

```
réflexion = 45
```

peut être parfaitement normale selon la surface.

Le programme peut afficher la valeur.

L'utilisateur doit parfois l'interpréter.

40. Vérifier les hypothèses

Chaque programme contient des hypothèses.

Par exemple :

```
if distance < 200:
```

suppose que :

```
200 mm
```

est une bonne distance d'arrêt.

Si le robot roule très vite, cette hypothèse peut être mauvaise.

Le code fonctionne, mais le choix du seuil est incorrect.

41. Déboguer avec des valeurs extrêmes

Si un comportement ne semble pas fonctionner, nous pouvons temporairement utiliser une valeur très évidente.

Exemple :

```
if distance < 1000:
```

Si le robot réagit maintenant, nous savons que la logique fonctionne probablement.

Le problème vient peut être du seuil initial.

42. Modifier une seule variable

Supposons que le suivi de ligne soit instable.

Nous avons :

```
vitesse = 300
```

```
kp = 4
```

```
kd = 3
```

Ne pas modifier les trois en même temps.

Tester d'abord :

```
vitesse = 200
```

sans changer le reste.

Observer.

Puis éventuellement modifier `kp`.

43. Pourquoi ?

Si nous changeons :

```
vitesse
```

```
kp
```

```
kd
```

```
hauteur du capteur
```

en même temps et que le robot fonctionne mieux, nous ne savons pas quelle modification est responsable.

Une bonne expérience modifie une variable à la fois.

44. Créer une hypothèse

Le débogage peut être vu comme une démarche scientifique.

Exemple :

```
Observation :
```

```
le robot perd la ligne dans les courbes.
```

Hypothèse :

```
la vitesse est trop élevée.
```

Test :

```
réduire uniquement la vitesse.
```

Résultat :

```
le robot ne perd plus la ligne.
```

Conclusion :

```
l'hypothèse était probablement correcte.
```

45. Exercice pratique 1 : programme cassé

Trouver les erreurs :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

vitesse = 300

if vitesse > 200

moteur.run(vitesse)
```

Correction :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

moteur = Motor(Port.A)

vitesse = 300

if vitesse > 200:

    moteur.run(vitesse)
```

46. Exercice pratique 2 : erreur logique

Programme :

```
distance_cm = 50

distance_mm = distance_cm / 10

robot.straight(distance_mm)
```

Question :

Pourquoi le robot n'avance t il pas de 50 cm ?

Correction :

```
distance_mm = distance_cm * 10
```

47. Exercice pratique 3 : mauvais seuil

Programme :

```
distance = capteur_ultrason.distance()

if distance < 20:

    robot.stop()
```

Si tu voulais arrêter le robot à :

```
20 cm
```

Le problème est l'unité.

Correction :

```
if distance < 200:
```

48. Exercice pratique 4 : erreur de comparaison

Incorrect :

```
if etat = "STOP":
```

Correct :

```
if etat == "STOP":
```

Rappel :

```
= attribue une valeur
== compare deux valeurs
```

49. Exercice pratique 5 : boucle infinie involontaire

```
compteur = 0

while compteur < 5:

    print(compteur)
```

Le compteur n'est jamais modifié.

La condition :

```
compteur < 5
```

reste donc toujours vraie.

Correction :

```
compteur = 0

while compteur < 5:

    print(compteur)

    compteur += 1
```

50. Exercice pratique 6 : machine à états bloquée

```
etat = "RECULER"

while True:

    if etat == "RECULER":

        robot.drive(-150, 0)
```

Le robot ne peut jamais quitter l'état.

Il faut ajouter une transition.

Par exemple :

```
if robot.distance() <= -150:

    robot.stop()

    etat = "TOURNER"
```

51. Exercice pratique 7 : PID incorrect

Trouver le problème :

```
d = (
    erreur
    - erreur
) * kd
```

Nous obtenons toujours :

```
0
```

Il faut utiliser :

```
d = (
    erreur
    - erreur_precedente
) * kd
```

52. Exercice pratique 8 : intégrale incorrecte

Programme :

```
somme_erreurs = erreur
```

dans chaque boucle.

Le programme remplace l'ancienne valeur au lieu de l'accumuler.

Correction :

```
somme_erreurs += erreur
```

53. Vérifier les valeurs avant les moteurs

Lorsqu'on teste un nouvel algorithme, il peut être intéressant de commencer sans déplacer le robot.

Par exemple pour le PID :

```
valeur = capteur_couleur.reflection()
```

```
erreur = cible - valeur
```

```
correction = erreur * kp
```

```
print(correction)
```

Déplacer manuellement le capteur.

Vérifier que le signe et la taille de la correction sont logiques.

Ensuite seulement activer :

```
robot.drive(vitesse, correction)
```

54. Cette technique améliore la sécurité

Tester d'abord les calculs permet d'éviter qu'un mauvais paramètre provoque :

```
une accélération soudaine
```

```
une rotation brutale
```

```
une collision
```

C'est particulièrement important avec des robots plus puissants.

55. Vérifier les limites

Une fonction `limiter()` peut protéger certains paramètres.

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

Puis :

```
correction = limiter(
    correction,
    -200,
    200
)
```

56. Ajouter des contrôles de cohérence

Exemple :

```
if vitesse < 0:

    print("Erreur vitesse")

    vitesse = 0
```

Ou :

```
if cible < 0 or cible > 100:

    print("Cible incorrecte")
```

Ces vérifications peuvent aider à détecter des paramètres impossibles.

57. Programme de diagnostic général

Nous pouvons créer un programme qui vérifie les principaux composants.

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import (
    Motor,
    TouchSensor,
    ColorSensor,
    UltrasonicSensor
)
from pybricks.parameters import Port
from pybricks.tools import wait


ev3 = EV3Brick()

moteur_gauche = Motor(Port.B)
```

```
moteur_droit = Motor(Port.C)

capteur_tactile = TouchSensor(Port.S1)
capteur_couleur = ColorSensor(Port.S3)
capteur_ultrason = UltrasonicSensor(Port.S4)


print("Test moteur gauche")

moteur_gauche.run(200)
wait(1000)
moteur_gauche.stop()


print("Test moteur droit")

moteur_droit.run(200)
wait(1000)
moteur_droit.stop()


print("Capteur tactile")
print(
    capteur_tactile.pressed()
)


print("Capteur couleur")
print(
    capteur_couleur.reflection()
)


print("Capteur ultrason")
print(
    capteur_ultrason.distance()
)


ev3.speaker.beep()

print("Diagnostic terminé")
```

58. Créer une fiche de diagnostic

Lorsqu'un problème apparaît, remplir :

Élément	Test	Résultat
Moteur gauche	Tourne seul	
Moteur droit	Tourne seul	
TouchSensor	True et False	
ColorSensor	Noir et blanc différents	
Ultrason	Distance varie	
DriveBase	Avance droit	
Calibration	Valeurs récentes	

Cela oblige à vérifier le robot méthodiquement.

59. Défi pratique : robot saboté

Le professeur peut préparer plusieurs programmes avec des erreurs.

Exemples :

```
mauvais port

mauvais signe

mauvaise unité

mauvaise indentation

condition incorrecte

kp trop élevé

seuil mal calibré

transition impossible
```

Chaque étudiant doit trouver et corriger les problèmes.

60. Niveau 1

Une seule erreur visible.

Exemple :

```
capteur = TouchSensor(Port.S4)
```

alors que le capteur est sur S1.

61. Niveau 2

Deux erreurs.

Par exemple :

```
distance = capteur_ultrason.distance()

if distance < 20:

robot.stop()
```

Tu dois identifier :

```
mauvaise unité

mauvaise indentation
```

62. Niveau 3

Erreur de logique.

```
while capteur_ultrason.distance() < 200:

    robot.drive(200, 0)
```

Si l'objectif est d'avancer tant que l'obstacle est loin, la condition est inversée.

Correction :

```
while capteur_ultrason.distance() >= 200:

    robot.drive(200, 0)
```

63. Niveau 4

Bug PID.

```
erreur_precedente = 0

while True:

    erreur = cible - capteur_couleur.reflection()

    erreur_precedente = erreur

    d = (
        erreur
        - erreur_precedente
    ) * kd
```

Ici, `erreur_precedente` est mise à jour trop tôt.

D vaut donc toujours :

```
0
```

Correction :

```
d = (
    erreur
    - erreur_precedente
) * kd

erreur_precedente = erreur
```

64. Niveau 5

Machine à états.

```
if etat == "SUIVRE":
    if distance < 200:
        etat = "RECULER"

elif etat == "RECULER":
    if distance < 200:
        etat = "SUIVRE"
```

Le robot peut alterner très rapidement entre les deux états.

Une meilleure stratégie serait d'utiliser des seuils différents ou une distance de recul précise.

65. Ne pas masquer un problème mécanique avec du code

Supposons que le robot tire fortement à gauche.

Une mauvaise réaction serait de modifier immédiatement :

```
moteur_gauche
```

ou la correction logicielle.

Mais il faut d'abord vérifier :

```
la roue gauche tourne librement
les deux roues sont identiques
les moteurs sont fixés
aucune pièce ne frotte
```

66. Le principe de la cause racine

Nous ne voulons pas seulement corriger le symptôme.

Nous voulons trouver :

```
la cause réelle du problème
```

Exemple :

```
Symptôme :
robot tourne légèrement à gauche
```

Cause possible 1 :

```
calibration DriveBase incorrecte
```

Cause possible 2 :

```
roue gauche qui frotte
```

Cause possible 3 :

```
diamètre des roues différent
```

Le diagnostic doit déterminer laquelle est correcte.

67. Méthode des cinq pourquoi

Une technique simple consiste à demander plusieurs fois :

```
Pourquoi ?
```

Exemple :

Pourquoi le robot s'est arrêté ?

Parce que l'état est devenu STOP.

Pourquoi l'état est devenu STOP ?

Parce que `distance < 200`.

Pourquoi la distance était inférieure à 200 ?

Parce que le capteur affichait 80.

Pourquoi affichait il 80 alors qu'il n'y avait aucun obstacle ?

Peut être parce qu'un câble est devant le capteur.

Nous sommes passés du symptôme à la cause.

68. Défi final : diagnostic complet

Le professeur fournit un robot ou un programme contenant plusieurs problèmes.

Tu dois :

1. écrire les symptômes
2. classer les problèmes
3. tester les composants
4. utiliser `print()`
5. identifier les causes
6. corriger une erreur à la fois
7. documenter les corrections

69. Rapport de diagnostic

Chaque étudiant complète :

```
Problème observé :

_____________________
Hypothèse :

_____________________
Test réalisé :

_____________________
Résultat :

_____________________
Cause trouvée :

_____________________
Correction :

_____________________
```

70. Questions de fin de cours

Question 1

Qu'est ce que le débogage ?

Question 2

Quelle différence existe entre une erreur de syntaxe et une erreur de logique ?

Question 3

À quoi sert `print()` pendant le développement ?

Question 4

Pourquoi tester séparément chaque capteur ?

Question 5

Pourquoi faut il vérifier les ports ?

Question 6

Pourquoi un programme correct peut il quand même donner un mauvais résultat ?

Question 7

Pourquoi modifier une seule variable à la fois ?

Question 8

Pourquoi est il utile d'isoler un problème ?

Question 9

Que signifie reproduire un bug ?

Question 10

Pourquoi un problème mécanique ne doit il pas être systématiquement corrigé avec du code ?

Correction

Réponse 1

Le débogage consiste à trouver, comprendre et corriger les problèmes d'un programme ou d'un système.

Réponse 2

Une erreur de syntaxe empêche Python de comprendre correctement le programme.

Une erreur de logique produit un résultat incorrect même si le programme fonctionne.

Réponse 3

À observer les valeurs internes du programme.

Réponse 4

Pour déterminer précisément quel composant fonctionne ou ne fonctionne pas.

Réponse 5

Parce que le programme doit correspondre au branchement réel.

Réponse 6

À cause par exemple d'une mauvaise calibration, d'un mauvais paramètre ou d'un problème mécanique.

Réponse 7

Pour pouvoir identifier l'effet réel de la modification.

Réponse 8

Parce qu'un petit problème isolé est généralement beaucoup plus simple à analyser qu'un programme complet.

Réponse 9

Pouvoir provoquer à nouveau le même problème dans les mêmes conditions.

Réponse 10

Parce que le code ne supprime pas forcément la véritable cause mécanique du problème.

Difficulté fréquente

Tu changes plusieurs valeurs au hasard jusqu'à ce que le programme fonctionne.

Même si tu finis par réussir, tu n'as pas réellement compris le problème.

Se demander systématiquement :

Quelle est mon hypothèse avant de modifier cette valeur ?

Autre difficulté fréquente

Se concentrer exclusivement sur le programme.

Un robot est un système :

```
logiciel

+

électronique

+

mécanique
```

Les trois parties doivent être vérifiées.

Cours 22 : Combiner plusieurs capteurs et gérer les priorités avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. utiliser plusieurs capteurs dans un même programme
2. lire plusieurs informations à chaque cycle
3. combiner des conditions
4. comprendre la notion de priorité
5. décider quel événement doit être traité en premier
6. combiner capteur tactile, ultrason et couleur
7. intégrer plusieurs capteurs dans une machine à états
8. créer des fonctions de lecture des capteurs
9. gérer des événements de sécurité
10. construire un robot autonome utilisant plusieurs sources d'information

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur tactile EV3
4. 1 capteur ultrason EV3
5. 1 capteur de couleur EV3
6. 2 roues identiques
7. 1 ordinateur avec Visual Studio Code
8. MicroPython installé sur l'EV3
9. une piste noire sur fond blanc
10. plusieurs obstacles
11. une zone rouge servant d'arrivée

1. Introduction

Jusqu'à présent, nous avons surtout étudié les capteurs séparément.

Le capteur tactile permet de détecter :

```
un contact
```

Le capteur ultrason permet de mesurer :

```
une distance
```

Le capteur de couleur permet de mesurer :

```
une couleur
```

```
ou
```

```
la lumière réfléchie
```

Un véritable robot autonome utilise souvent plusieurs capteurs en même temps.

Par exemple :

```
Suivre une ligne

↓

Surveiller les obstacles

↓

Détecter une collision

↓

Détecter l'arrivée
```

Le programme doit donc analyser plusieurs informations continuellement.

2. Configuration du robot

Pour cette leçon, nous utiliserons :

```
Port B = moteur gauche

Port C = moteur droit

Port S1 = capteur tactile

Port S3 = capteur de couleur

Port S4 = capteur ultrason
```

3. Initialiser tous les composants

```
from pybricks.hubs import EV3Brick

from pybricks.ev3devices import (
    Motor,
    TouchSensor,
    ColorSensor,
    UltrasonicSensor
)

from pybricks.parameters import (
    Port,
    Color
)

from pybricks.robotics import DriveBase


ev3 = EV3Brick()

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_tactile = TouchSensor(Port.S1)
capteur_couleur = ColorSensor(Port.S3)
capteur_ultrason = UltrasonicSensor(Port.S4)
```

```
robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

Nous avons maintenant accès à plusieurs informations sur l'environnement.

4. Lire plusieurs capteurs

Nous pouvons écrire :

```
distance = capteur_ultrason.distance()

couleur = capteur_couleur.color()

reflection = capteur_couleur.reflection()

contact = capteur_tactile.pressed()
```

Chaque variable représente une information différente.

5. Afficher toutes les informations

```
from pybricks.tools import wait

while True:

    distance = capteur_ultrason.distance()

    reflection = capteur_couleur.reflection()

    contact = capteur_tactile.pressed()

    print("Distance")
    print(distance)

    print("Reflection")
    print(reflection)

    print("Contact")
    print(contact)

    wait(200)
```

Ce programme permet d'observer plusieurs capteurs simultanément.

6. Exercice pratique 1

Déplacer un obstacle devant le robot.

Appuyer sur le capteur tactile.

Déplacer le robot entre une surface noire et blanche.

Observer les différentes valeurs.

Compléter :

Information	Valeur observée
Distance obstacle proche	
Distance obstacle loin	
Réflexion noir	
Réflexion blanc	
Capteur tactile libre	
Capteur tactile appuyé	

7. Le problème des décisions simultanées

Imaginons que le robot détecte au même moment :

```
une ligne noire
```

et :

```
un obstacle à 10 cm
```

Que doit-il faire ?

Continuer à suivre la ligne ?

Ou éviter l'obstacle ?

Il faut définir une priorité.

8. Qu'est-ce qu'une priorité ?

Une priorité indique quelle action est la plus importante lorsqu'il existe plusieurs événements en même temps.

Par exemple :

```
Priorité 1

ARRÊT D'URGENCE
Priorité 2

COLLISION
Priorité 3

OBSTACLE
Priorité 4

SUIVI DE LIGNE
```

Le robot traite d'abord l'événement le plus important.

9. Exemple simple

Nous voulons :

```
Si tactile appuyé

STOP

Sinon si obstacle proche

RECULER

Sinon

AVANCER
```

En Python :

```
if capteur_tactile.pressed():

    robot.stop()

elif capteur_ultrason.distance() < 200:

    robot.drive(-150, 0)

else:

    robot.drive(200, 0)
```

10. Pourquoi utiliser `elif` ?

Avec :

```
if
elif
else
```

une seule branche est exécutée.

Dès qu'une condition est vraie, les suivantes ne sont plus testées pour ce cycle.

Cela permet de définir une priorité naturelle.

11. Comparaison avec plusieurs `if`

Considérons :

```
if capteur_tactile.pressed():

    robot.stop()


if capteur_ultrason.distance() < 200:

    robot.drive(-150, 0)
```

Si les deux conditions sont vraies, le programme exécute :

```
STOP
```

puis immédiatement :

```
RECULER
```

Le deuxième ordre peut donc annuler le premier.

Ce n'est peut-être pas ce que nous voulons.

12. Avec `elif`

```
if capteur_tactile.pressed():

    robot.stop()

elif capteur_ultrason.distance() < 200:

    robot.drive(-150, 0)
```

Si le tactile est appuyé, le deuxième cas n'est pas exécuté.

Le tactile possède donc une priorité plus élevée.

13. Exercice pratique 2

Créer :

```
Priorité 1

capteur tactile
↓
STOP
Priorité 2

distance inférieure à 25 cm
↓
RECULER
Priorité 3

sinon
↓
AVANCER
```

Solution :

```
if capteur_tactile.pressed():

    robot.stop()

elif capteur_ultrason.distance() < 250:

    robot.drive(-150, 0)

else:

    robot.drive(200, 0)
```

14. Ajouter la couleur rouge

Nous voulons maintenant une zone d'arrivée.

Si le robot détecte :

```
Color.RED
```

il doit s'arrêter définitivement.

La priorité peut devenir :

```
1. ROUGE

2. CONTACT

3. OBSTACLE

4. COMPORTEMENT NORMAL
```

15. Exemple

```
couleur = capteur_couleur.color()
contact = capteur_tactile.pressed()
distance = capteur_ultrason.distance()

if couleur == Color.RED:

    robot.stop()

elif contact:

    robot.stop()

elif distance < 200:

    robot.drive(-150, 0)

else:

    robot.drive(200, 0)
```

16. Mais le rouge doit arrêter définitivement le programme

Dans l'exemple précédent, si le robot quitte ensuite la zone rouge, il pourrait recommencer à avancer.

Nous pouvons utiliser :

```
break
```

Exemple :

```
while True:
    couleur = capteur_couleur.color()
    if couleur == Color.RED:
        robot.stop()
        break
    robot.drive(200, 0)
```

17. Priorité de sécurité

Dans beaucoup de systèmes, certaines conditions doivent être vérifiées avant tout le reste.

Par exemple :

```
STOP D'URGENCE
↓
priorité maximale
```

Même si le robot est :

```
en train de suivre la ligne
en train d'éviter un obstacle
en train de chercher la ligne
```

une urgence doit pouvoir interrompre son comportement.

18. Application aux machines à états

Nous avons vu précédemment :

```
etat = "SUIVRE"
```

Puis :

```
if etat == "SUIVRE":
```

Mais nous pouvons vérifier les événements prioritaires avant la machine à états.

```
while True:
    if capteur_tactile.pressed():
        etat = "STOP"
    elif capteur_couleur.color() == Color.RED:
        etat = "STOP"
    # Machine à états ensuite
```

19. Structure générale

```
while True:

    # Événements prioritaires

    if urgence:

        etat = "STOP"


    # Machine à états

    if etat == "SUIVRE":

        # comportement

    elif etat == "EVITER":

        # comportement

    elif etat == "STOP":

        robot.stop()
        break
```

Cette structure permet de séparer :

```
les événements globaux
```

et :

```
le comportement normal
```

20. Créer des fonctions de lecture

Nous pouvons écrire :

```
def obstacle_proche(limite):

    return (
        capteur_ultrason.distance()
        < limite
    )
```

Puis :

```
if obstacle_proche(200):
```

Le programme devient plus lisible.

21. Fonction de collision

```
def collision():

    return capteur_tactile.pressed()
```

Puis :

```
if collision():

    robot.stop()
```

22. Fonction de détection du rouge

```
def zone_rouge():

    return (
        capteur_couleur.color()
        == Color.RED
    )
```

Puis :

```
if zone_rouge():

    robot.stop()
```

23. Pourquoi ces fonctions sont-elles intéressantes ?

Comparer :

```
if (
    capteur_couleur.color()
    == Color.RED
):
```

avec :

```
if zone_rouge():
```

La deuxième version est plus facile à comprendre.

Nous cachons la manière technique de détecter la couleur derrière une fonction au nom clair.

24. Créer une fonction de sécurité

Nous pouvons même créer :

```
def danger():

    if collision():
        return True

    if obstacle_proche(100):
        return True

    return False
```

Puis :

```
if danger():

    robot.stop()
```

25. Combiner avec `or`

La même logique peut être écrite :

```
def danger():

    return (
        collision()
        or obstacle_proche(100)
    )
```

Cela signifie :

```
collision

OU

obstacle extrêmement proche
```

26. Combiner avec `and`

Supposons que nous voulions détecter une situation particulière :

```
Obstacle proche

ET

capteur tactile appuyé
```

Nous pouvons écrire :

```
if (
    capteur_ultrason.distance() < 200
    and capteur_tactile.pressed()
):

    print("Collision confirmée")
```

27. Exercice de logique

Que signifie :

```
if (
    distance < 200
    and couleur == Color.RED
):
```

Réponse :

Les deux conditions doivent être vraies en même temps.

28. Que signifie `or` ?

```
if (
    capteur_tactile.pressed()
    or couleur == Color.RED
):
```

Il suffit qu'une des deux conditions soit vraie.

29. Que signifie `not` ?

```
if not capteur_tactile.pressed():
```

signifie :

```
si le capteur tactile n'est pas appuyé
```

30. Exercice pratique 3

Le robot doit avancer uniquement si :

```
aucun contact
```

ET :

```
obstacle à plus de 30 cm
```

Une solution :

```
if (
    not capteur_tactile.pressed()
    and capteur_ultrason.distance() > 300
):

    robot.drive(200, 0)

else:

    robot.stop()
```

31. Ajouter le suivi de ligne

Notre comportement normal sera maintenant :

```
SUIVRE LA LIGNE
```

Nous pouvons créer :

```
def suivi_cycle():

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(
        vitesse,
        correction
    )
```

32. Priorités pendant le suivi

Nous voulons :

```
Rouge

↓

STOP
```

sinon :

```
Collision

↓

RECULER
```

sinon :

```
Obstacle proche

↓

EVITER
```

sinon :

```
SUIVRE
```

33. Première version

```
while True:

    if zone_rouge():

        robot.stop()
        break

    elif collision():

        robot.drive(-150, 0)

    elif obstacle_proche(200):

        robot.stop()

    else:

        suivi_cycle()
```

Le programme est déjà capable d'utiliser trois capteurs.

34. Limite de ce programme

L'évitement d'obstacle nécessite plusieurs étapes.

Simplement écrire :

```
robot.stop()
```

ne suffit pas.

Nous allons donc combiner les priorités avec une machine à états.

35. États du robot

Nous utiliserons :

```
SUIVRE

RECULER

TOURNER

CONTOURNER

CHERCHER

STOP
```

36. Priorité globale

La zone rouge sera vérifiée dans tous les états.

```
if zone_rouge():
    etat = "STOP"
```

Le capteur tactile peut également être considéré comme une sécurité.

```
if collision():
    robot.stop()
    robot.reset()

    etat = "RECULER"
```

37. Attention à la priorité entre rouge et tactile

Supposons que :

```
rouge détecté
```

et :

```
tactile appuyé
```

soient vrais en même temps.

Si le rouge signifie :

```
FIN IMMÉDIATE
```

il doit être vérifié en premier.

```
if zone_rouge():
    etat = "STOP"

elif collision():
    etat = "RECULER"
```

38. Concevoir les priorités avant de programmer

Avant d'écrire le code, créer un tableau.

Priorité	Événement	Réaction
1	Zone rouge	STOP
2	Collision	RECULER
3	Obstacle proche	EVITER
4	Ligne perdue	CHERCHER
5	Aucun problème	SUIVRE

Cette étape évite beaucoup d'erreurs de logique.

39. Pourquoi la ligne perdue est-elle moins prioritaire qu'un obstacle ?

Si le robot :

```
perd légèrement la ligne
```

mais qu'un obstacle se trouve juste devant lui, éviter la collision est plus urgent.

Nous devons donc traiter :

```
OBSTACLE
```

avant :

```
RECHERCHE DE LIGNE
```

40. Détection de ligne perdue

Nous pouvons utiliser :

```
limite_perte = 75
compteur_perte = 0
```

Puis :

```
if valeur > limite_perte:
    compteur_perte += 1

else:
    compteur_perte = 0
```

Si :

```
compteur_perte > 20
```

le robot considère que la ligne est réellement perdue.

41. Pourquoi utiliser un compteur ?

Une seule mesure claire peut être normale.

Par exemple :

```
petite interruption de ligne

variation de lumière

surface imparfaite
```

Nous ne voulons pas immédiatement quitter l'état `SUIVRE`.

Le compteur demande que le problème persiste plusieurs cycles.

42. Fonction de suivi avec information

Nous pouvons créer une fonction qui retourne la valeur mesurée.

```
def suivi_cycle():

    valeur = capteur_couleur.reflection()

    erreur = cible - valeur

    correction = erreur * kp

    robot.drive(
        vitesse,
        correction
    )

    return valeur
```

Puis :

```
valeur = suivi_cycle()
```

Nous pouvons ensuite utiliser cette valeur pour détecter une éventuelle perte de ligne.

43. État SUIVRE

```
if etat == "SUIVRE":

    valeur = suivi_cycle()

    if obstacle_proche(200):

        robot.stop()
        robot.reset()

        etat = "RECULER"
```

Mais souvenons nous que les événements prioritaires peuvent déjà avoir été traités avant.

44. État RECULER

```
elif etat == "RECULER":

    robot.drive(-150, 0)

    if robot.distance() <= -150:

        robot.stop()
        robot.reset()

        etat = "TOURNER"
```

45. État TOURNER

```
elif etat == "TOURNER":

    robot.drive(0, 90)

    if robot.angle() >= 90:

        robot.stop()
        robot.reset()

        etat = "CONTOURNER"
```

46. État CONTOURNER

Dans cet état, le robot avance à côté de l'obstacle.

Par exemple :

```
elif etat == "CONTOURNER":

    robot.drive(150, 0)

    if robot.distance() >= 300:

        robot.stop()

        etat = "CHERCHER"
```

Les valeurs exactes dépendront du terrain.

47. État CHERCHER

```
elif etat == "CHERCHER":

    robot.drive(100, -40)

    if (
        capteur_couleur.reflection()
        < cible
    ):

        robot.stop()

        etat = "SUIVRE"
```

Le robot tourne progressivement jusqu'à retrouver une zone sombre.

48. État STOP

```
elif etat == "STOP":

    robot.stop()

    break
```

Le programme se termine.

49. Structure complète

```
etat = "SUIVRE"

while True:

    # PRIORITÉS GLOBALES

    if zone_rouge():
        etat = "STOP"

    elif collision():
        if etat != "STOP":
            robot.stop()
            robot.reset()
            etat = "RECULER"

    # MACHINE À ÉTATS

    if etat == "SUIVRE":
        suivi_cycle()

        if obstacle_proche(200):
            robot.stop()
            robot.reset()
            etat = "RECULER"

    elif etat == "RECULER":
        robot.drive(-150, 0)

        if robot.distance() <= -150:
            robot.stop()
            robot.reset()
            etat = "TOURNER"

    elif etat == "TOURNER":
        robot.drive(0, 90)

        if robot.angle() >= 90:
            robot.stop()
            robot.reset()
            etat = "CONTOURNER"

    elif etat == "CONTOURNER":

        robot.drive(150, 0)

        if robot.distance() >= 300:
            robot.stop()
            etat = "CHERCHER"

    elif etat == "CHERCHER":
        robot.drive(100, -40)

        if (
            capteur_couleur.reflection()
            < cible
        ):

            robot.stop()
            etat = "SUIVRE"

    elif etat == "STOP":
        robot.stop()
        break
```

50. Pourquoi ce programme est-il intéressant ?

Il combine :

`3 capteurs`

avec :

`6 états`

et plusieurs priorités.

Pourtant, chaque partie possède un rôle clair.

Nous pouvons lire séparément :

`les priorités`

puis :

`les différents états`

51. Déboguer un programme multi capteurs

Plus nous utilisons de capteurs, plus il est important de connaître les valeurs réellement lues.

Nous pouvons temporairement afficher :

```
print("Etat")
print(etat)

print("Distance")
print(capteur_ultrason.distance())

print("Reflection")
print(capteur_couleur.reflection())

print("Contact")
print(capteur_tactile.pressed())
```

52. Attention à trop de `print()`

Afficher énormément d'informations à chaque cycle peut ralentir le programme.

Pour le diagnostic, c'est utile.

Pour la version finale, nous pouvons retirer les affichages inutiles.

53. Programme de diagnostic des trois capteurs

```
from pybricks.tools import wait

while True:
    print("TACTILE")
    print(
        capteur_tactile.pressed()
    )

    print("ULTRASON")
    print(
        capteur_ultrason.distance()
    )

    print("COULEUR")
    print(
        capteur_couleur.reflection()
    )
    wait(300)
```

Avant de lancer un projet complexe, il est intéressant de vérifier que les trois capteurs fonctionnent correctement.

54. Les capteurs peuvent se compléter

L'ultrason permet :

```
d'anticiper un obstacle
```

Le tactile permet :

```
de confirmer une collision
```

Le capteur de couleur permet :

```
de suivre la piste
```

Nous pouvons donc utiliser plusieurs capteurs pour rendre le comportement plus robuste.

55. Exemple de sécurité supplémentaire

Le robot commence normalement à éviter un obstacle sous :

```
250 mm
```

Mais si malgré cela le tactile est appuyé :

```
collision réelle
```

nous pouvons forcer :

```
STOP
puis recul
```

Le tactile joue alors le rôle de sécurité supplémentaire.

56. Redondance des capteurs

Lorsque plusieurs capteurs peuvent détecter un même problème, on parle parfois de redondance.

Exemple :

```
Ultrason
↓
Obstacle détecté avant contact
```

Mais si l'ultrason ne détecte pas correctement :

```
Tactile
↓
Collision détectée
```

Le deuxième capteur constitue une protection supplémentaire.

57. Exercice pratique 4

Créer un robot qui avance.

Comportement :

```
Distance supérieure à 300 mm
↓
AVANCER
Distance inférieure à 300 mm
↓
RALENTIR
Distance inférieure à 150 mm
↓
STOP
Tactile appuyé
↓
RECULER
```

Le tactile doit avoir priorité sur la distance.

58. Solution possible

```
while True:
    distance = (
        capteur_ultrason.distance()
    )

    if capteur_tactile.pressed():
        robot.drive(-150, 0)

    elif distance < 150:
        robot.stop()

    elif distance < 300:
        robot.drive(100, 0)

    else:
        robot.drive(300, 0)
```

59. Exercice pratique 5

Ajouter :

```
Zone rouge
↓
arrêt définitif
```

Elle doit avoir la priorité maximale.

Solution :

```
while True:

    couleur = capteur_couleur.color()

    distance = (
        capteur_ultrason.distance()
    )

    if couleur == Color.RED:
        robot.stop()
        break

    elif capteur_tactile.pressed():
        robot.drive(-150, 0)

    elif distance < 150:
        robot.stop()

    elif distance < 300:
        robot.drive(100, 0)

    else:
        robot.drive(300, 0)
```

60. Exercice pratique 6 : priorité incorrecte

Pourquoi ce programme peut il être dangereux ?

```
if distance > 300:
    robot.drive(300, 0)

elif couleur == Color.RED:
    robot.stop()
```

Si le robot est sur une zone rouge mais que :

```
distance > 300
```

la première condition est vraie.

Le robot continue donc à avancer.

Il faut placer l'arrêt prioritaire avant le comportement normal.

61. Correction

```
if couleur == Color.RED:

    robot.stop()

elif distance > 300:

    robot.drive(300, 0)
```

L'ordre des conditions fait donc partie de la logique du robot.

62. Créer une fonction de décision

Nous pouvons imaginer :

```
def evenement_prioritaire():

    if zone_rouge():
        return "STOP"

    if collision():
        return "COLLISION"

    if obstacle_proche(200):
        return "OBSTACLE"

    return "AUCUN"
```

Puis :

```
evenement = evenement_prioritaire()
```

63. Utiliser le résultat

```
if evenement == "STOP":

    etat = "STOP"

elif evenement == "COLLISION":

    etat = "RECULER"

elif evenement == "OBSTACLE":

    etat = "RECULER"
```

Cela sépare encore davantage :

```
la détection
```

de :

```
la réaction
```

64. Pourquoi séparer les responsabilités ?

Une fonction peut être responsable de :

```
lire les capteurs
```

Une autre de :

```
prendre une décision
```

Une autre de :

```
commander les moteurs
```

Un programme bien structuré est plus facile à maintenir.

65. Exemple d’architecture

```
CAPTEURS
↓
FONCTIONS DE LECTURE
↓
DÉCISION
↓
ÉTAT
↓
ACTION
↓
MOTEURS
```

Cette architecture ressemble déjà à celle de systèmes robotiques plus avancés.

66. Défi pratique 7 : trois événements

Créer un robot qui suit une ligne mais doit gérer :

```
Obstacle
```

```
Zone rouge
```

```
Collision
```

Priorités imposées :

```
1. Rouge
```

```
2. Collision
```

```
3. Obstacle
```

```
4. Ligne
```

Tu dois expliquer pourquoi cet ordre a été choisi.

67. Défi pratique 8 : compteur d’obstacles

Ajouter :

```
nombre_obstacles = 0
```

Chaque évitement réussi augmente le compteur.

Après :

```
3 obstacles
```

le robot doit produire un son.

Après :

```
5 obstacles
```

il doit s’arrêter.

68. Exemple

```
nombre_obstacles = 0
```

Lors de la détection :

```
nombre_obstacles += 1

if nombre_obstacles == 3:
    ev3.speaker.beep()

if nombre_obstacles >= 5:
    etat = "STOP"
```

69. Défi pratique 9 : priorité dynamique

Créer une règle :

```
Avant 3 obstacles
↓
comportement normal
```

Après 3 obstacles :

```
vitesse réduite
```

Par exemple :

```
if nombre_obstacles >= 3:
    vitesse = 100

else:
    vitesse = 200
```

Le comportement du robot dépend maintenant aussi de son historique.

70. Défi final : mission autonome multi capteurs

Créer une piste comportant :

`une ligne noire`

`deux ou plusieurs obstacles`

`une zone rouge d'arrivée`

Le robot doit :

1. suivre la ligne
2. anticiper les obstacles avec l'ultrason
3. utiliser le tactile comme sécurité en cas de collision
4. éviter l'obstacle
5. retrouver la ligne
6. reprendre le suivi
7. compter les obstacles
8. s'arrêter définitivement sur la zone rouge

71. Contraintes du programme

Le programme doit utiliser :

1. au moins trois capteurs
2. une machine à états
3. au moins quatre états
4. plusieurs fonctions
5. `if`
6. `elif`
7. `and`, `or` ou `not`
8. au moins une priorité globale
9. un compteur
10. un état `STOP`

72. Exemple de structure finale

```
etat = "SUIVRE"

nombre_obstacles = 0

while True:

    # Priorité 1
    if zone_rouge():

        etat = "STOP"

    # Priorité 2
    elif collision():

        robot.stop()
        robot.reset()

        etat = "RECULER"
```

```
# Machine à états
if etat == "SUIVRE":

    suivi_cycle()

    if obstacle_proche(200):

        nombre_obstacles += 1

        robot.stop()
        robot.reset()

        etat = "RECULER"


elif etat == "RECULER":

    robot.drive(-150, 0)

    if robot.distance() <= -150:

        robot.stop()
        robot.reset()

        etat = "TOURNER"


elif etat == "TOURNER":

    robot.drive(0, 90)

    if robot.angle() >= 90:

        robot.stop()
        robot.reset()

        etat = "CHERCHER"


elif etat == "CHERCHER":

    robot.drive(100, -40)

    if (
        capteur_couleur.reflection()
        < cible
    ):

        robot.stop()

        etat = "SUIVRE"


elif etat == "STOP":

    robot.stop()

    break
```

73. Questions de fin de cours

Question 1

Pourquoi utiliser plusieurs capteurs ?

Question 2

Qu'est ce qu'une priorité ?

Question 3

Pourquoi l'ordre des `if` et `elif` peut il être important ?

Question 4

Quelle différence existe entre plusieurs `if` et une structure `if`, `elif`, `else` ?

Question 5

Que signifie `and` ?

Question 6

Que signifie `or` ?

Question 7

Que signifie `not` ?

Question 8

Pourquoi un capteur tactile peut il être utile même si nous possédons déjà un capteur ultrason ?

Question 9

Pourquoi vérifier certains événements avant la machine à états ?

Question 10

Pourquoi créer des fonctions comme `collision()` ou `obstacle_proche()` ?

Correction

Réponse 1

Parce que chaque capteur fournit des informations différentes et qu'ils peuvent se compléter.

Réponse 2

Une priorité indique quel événement doit être traité en premier lorsque plusieurs situations se produisent.

Réponse 3

Parce qu'avec `elif`, le premier cas vrai empêche les suivants d'être exécutés.

Réponse 4

Plusieurs `if` peuvent tous être exécutés pendant le même cycle.

Avec `if`, `elif`, `else`, une seule branche est sélectionnée.

Réponse 5

Les deux conditions doivent être vraies.

Réponse 6

Au moins une des conditions doit être vraie.

Réponse 7

Il inverse une valeur booléenne ou une condition.

Réponse 8

Parce qu'il peut détecter une collision réelle si l'ultrason n'a pas détecté correctement l'obstacle.

Réponse 9

Parce que certains événements comme un arrêt d'urgence doivent pouvoir interrompre n'importe quel comportement.

Réponse 10

Pour rendre le programme plus clair et séparer la manière de détecter un événement de la logique générale.

Difficulté fréquente

Lire plusieurs fois le même capteur dans différents endroits du cycle.

Pour certains programmes, il est plus simple de lire une fois :

```
distance = capteur_ultrason.distance()
contact = capteur_tactile.pressed()
valeur = capteur_couleur.reflection()
```

puis d'utiliser ces variables dans les conditions.

Cela permet aussi de savoir que toutes les décisions utilisent les valeurs du même cycle.

Difficulté fréquente

Donner trop de responsabilités à un seul capteur.

Par exemple :

```
Ultrason = obstacle
```

mais il n'est pas forcément fiable dans toutes les situations.

Le tactile peut servir de sécurité supplémentaire.

Cette complémentarité est une idée importante en robotique.

Cours 23 : Le gyroscope et les rotations précises avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. comprendre le rôle d'un gyroscope
2. connecter et initialiser un `GyroSensor`
3. mesurer un angle
4. remettre l'angle à zéro
5. effectuer une rotation contrôlée
6. comparer une rotation avec `DriveBase` et une rotation mesurée par gyroscope
7. maintenir une direction
8. corriger une dérive
9. intégrer le gyroscope dans une machine à états
10. préparer le robot au projet final autonome

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 gyroscope EV3
4. 2 roues identiques
5. 1 ordinateur avec Visual Studio Code
6. MicroPython installé sur l'EV3
7. du ruban adhésif pour marquer des angles au sol
8. une règle
9. un rapporteur si disponible
10. une piste de test

1. Introduction

Nous savons déjà faire tourner le robot avec :

```
robot.turn(90)
```

Cette commande est pratique.

Mais elle repose principalement sur les moteurs et les dimensions données à `DriveBase`.

Par exemple :

```
wheel_diameter=56
```

et :

```
axle_track=114
```

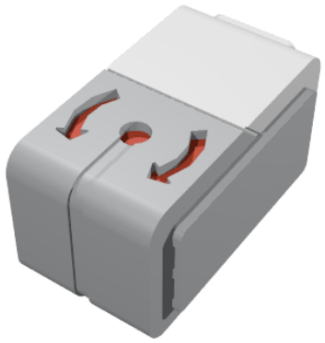
Si ces valeurs sont légèrement incorrectes, la rotation peut également être légèrement incorrecte.

Aujourd'hui, nous allons ajouter un capteur capable de mesurer directement la rotation du robot.

2. Le gyroscope

Un gyroscope mesure les rotations.

Pour notre robot, il peut notamment nous indiquer :



```
0°

30°

90°

180°

moins 90°
```

Nous pouvons donc demander au robot :

```
Tourne
↓
mesure ton angle
↓
continue
↓
90° atteint
↓
STOP
```

3. Pourquoi est-ce intéressant ?

Avec uniquement les moteurs, nous estimons la rotation à partir du mouvement des roues.

Avec le gyroscope, nous mesurons directement l'orientation du robot.

Cela permet de détecter certaines erreurs comme :

```
roue qui patine

rotation trop courte

rotation trop longue

déviation pendant un déplacement
```

4. Connexion

Pour cette leçon, nous allons utiliser :

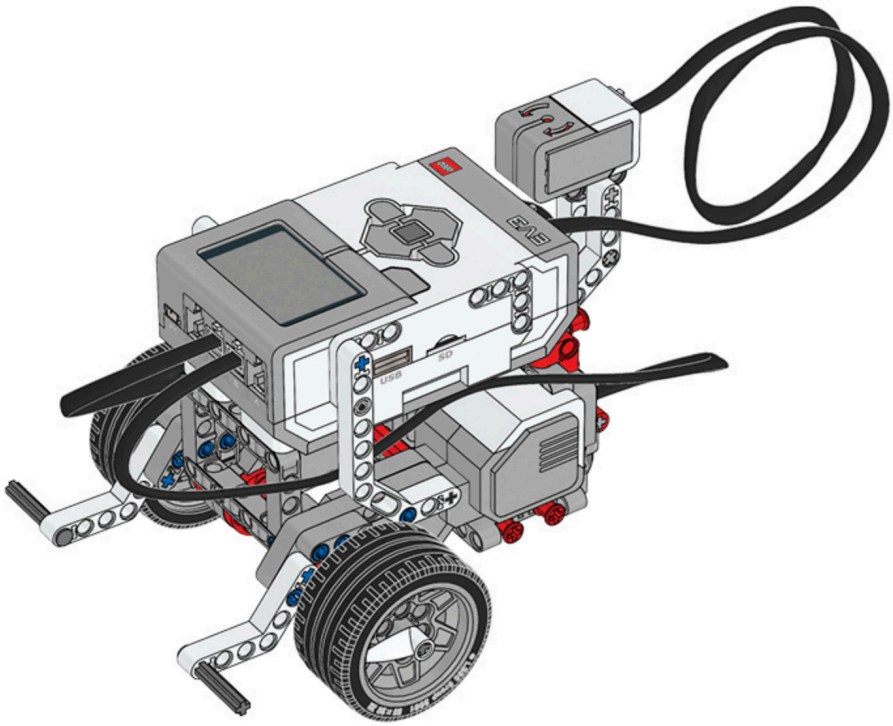
```
Port S2 = gyroscope
```

Configuration :

```
Port B = moteur gauche

Port C = moteur droit

Port S2 = gyroscope
```



5. Importer le gyroscope

Nous devons importer :

```
from pybricks.ev3devices import GyroSensor
```

Puis :

```
from pybricks.parameters import Port
```

Initialisation :

```
gyro = GyroSensor(Port.S2)
```

6. Premier programme

```
from pybricks.ev3devices import GyroSensor
from pybricks.parameters import Port
from pybricks.tools import wait

gyro = GyroSensor(Port.S2)

while True:

    print(gyro.angle())

    wait(200)
```

Tourner doucement le robot à la main.

Observer les valeurs.

7. Comprendre le signe

Selon le sens de rotation :

```
l’angle augmente
```

ou :

```
l’angle diminue
```

Par exemple :

```
0°
30°
60°
90°
```

dans un sens.

Et :

```
0°
moins 30°
moins 60°
moins 90°
```

dans l’autre.

Le sens exact dépend du montage et de l’orientation du capteur.

8. Exercice pratique 1

Placer le robot à :

```
0°
```

Tourner à la main d’environ :

```
90°
```

Noter la valeur.

Puis :

```
180°
```

Puis revenir à la position initiale.

Position approximative	Mesure
0°	
90°	
180°	
Retour	

9. Remettre l’angle à zéro

Avant une mesure, nous voulons souvent considérer la position actuelle comme :

```
0°
```

Nous pouvons utiliser :

```
gyro.reset_angle(0)
```

Puis :

```
print(gyro.angle())
```

10. Exemple

```
gyro.reset_angle(0)
while True:
    print(gyro.angle())
    wait(200)
```

La position du robot au moment du reset devient la nouvelle référence.

11. Pourquoi remettre à zéro ?

Supposons que le gyroscope indique déjà :

```
37°
```

avant notre virage.

Si nous voulons mesurer un nouveau virage de 90 degrés, il est plus simple de faire :

```
gyro.reset_angle(0)
```

Puis d’attendre :

```
90°
```

12. Premier virage contrôlé

Initialisons le robot.

```
from pybricks.ev3devices import (
    Motor,
    GyroSensor
)

from pybricks.parameters import Port

from pybricks.robotics import DriveBase


moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

gyro = GyroSensor(Port.S2)

robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

Puis :

```
gyro.reset_angle(0)

robot.drive(0, 80)

while gyro.angle() < 90:
    pass

robot.stop()
```

Le robot tourne jusqu'à atteindre environ 90 degrés.

13. Comprendre `robot.drive(0, 80)`

Le premier paramètre vaut :

```
0
```

Le robot n'avance donc pas.

Le deuxième paramètre demande une rotation.

```
robot.drive(0, 80)
```

produit donc une rotation sur place.

14. Attention au sens

Si l'angle diminue pendant cette rotation :

```
0
```

```
moins 10
```

```
moins 20
```

alors cette condition :

```
while gyro.angle() < 90:
```

restera vraie.

Il faut adapter le programme au sens réel du gyroscope.

Par exemple :

```
robot.drive(0, -80)

while gyro.angle() > -90:
    pass
```

15. Exercice pratique 2

Créer une rotation d'environ :

```
45°
```

Puis :

```
90°
```

Puis :

```
180°
```

Mesurer l'erreur réelle avec un repère au sol.

16. Créer une fonction

Nous pouvons créer :

```
def tourner_gyro(angle):

    gyro.reset_angle(0)

    robot.drive(0, 80)

    while gyro.angle() < angle:
        pass

    robot.stop()
```

Puis :

```
tourner_gyro(90)
```

17. Limite de cette fonction

Cette version fonctionne seulement pour un sens de rotation et des angles positifs.

Nous voulons maintenant gérer :

```
90
```

et :

```
moins 90
```

18. Fonction avec sens automatique

```
def tourner_gyro(angle):

    gyro.reset_angle(0)

    if angle > 0:
        robot.drive(0, 80)
        while gyro.angle() < angle:
            pass

    else:
        robot.drive(0, -80)
        while gyro.angle() > angle:
            pass

    robot.stop()
```

Nous pouvons maintenant utiliser :

```
tourner_gyro(90)
```

ou :

```
tourner_gyro(-90)
```

19. Exercice pratique 3

Créer :

```
tourner_gyro(180)
```

puis :

```
tourner_gyro(-180)
```

Observer si le robot revient approximativement dans la même orientation.

20. Pourquoi le robot peut dépasser l'angle ?

Supposons que nous arrêtions lorsque :

```
angle = 90°
```

Le robot possède encore de l'inertie.

Il peut finir à :

```
92°
```

```
94°
```

```
96°
```

Plus la vitesse de rotation est élevée, plus ce dépassement peut être important.

21. Expérience sur la vitesse

Tester :

```
vitesse_rotation = 40
```

puis :

```
80
```

puis :

```
150
```

Vitesse de rotation	Angle demandé	Angle final
40	90	
80	90	
150	90	

22. Ralentir à l'approche de la cible

Nous pouvons améliorer la rotation.

```
gyro.reset_angle(0)

while gyro.angle() < 90:
    angle_restant = 90 - gyro.angle()

    if angle_restant < 20:
        robot.drive(0, 30)

    else:
        robot.drive(0, 80)

robot.stop()
```

Le robot ralentit lorsqu'il approche de 90 degrés.

23. Calculer l'erreur angulaire

Comme pour le suivi de ligne :

```
erreur = angle_cible - angle_mesure
```

Exemple :

```
cible = 90

mesure = 70
```

Alors :

```
erreur = 20
```

24. Utiliser une correction proportionnelle

Nous pouvons écrire :

```
correction = erreur * kp
```

Puis utiliser la correction comme vitesse de rotation.

```
robot.drive(0, correction)
```

Le robot tourne rapidement quand il est loin de la cible et ralentit lorsqu'il s'en approche.

25. Premier contrôleur proportionnel pour l’angle

```
gyro.reset_angle(0)

angle_cible = 90

kp = 2

while True:

    angle = gyro.angle()

    erreur = angle_cible - angle

    correction = erreur * kp

    robot.drive(0, correction)

    if abs(erreur) < 2:
        break

robot.stop()
```

26. Pourquoi utiliser une tolérance ?

Il est possible que le gyroscope mesure :

```
89°

91°

90°

89°
```

Chercher exactement :

```
90
```

n’est pas toujours nécessaire.

Nous pouvons accepter :

```
plus ou moins 2°
```

Avec :

```
if abs(erreur) < 2:
```

27. Limiter la correction

Si :

```
erreur = 180

kp = 5
```

nous obtenons :

```
correction = 900
```

Cela peut être beaucoup trop rapide.

Nous pouvons limiter :

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

Puis :

```
correction = limiter(
    erreur * kp,
    -150,
    150
)
```

28. Fonction précise

```
def tourner_precis(angle_cible):

    gyro.reset_angle(0)
    kp = 2

    while True:
        angle = gyro.angle()
        erreur = (
            angle_cible
            - angle
        )

        if abs(erreur) < 2:
            break

        correction = erreur * kp
        correction = limiter(
            correction,
            -150,
            150
        )

        robot.drive(
            0,
            correction
        )

    robot.stop()
```

Puis :

```
tourner_precis(90)
```

29. Exercice pratique 4

Tester :

```
tourner_precis(90)
```

cinq fois.

Noter :

Essai	Angle final
1	
2	
3	
4	
5	

Calculer l’erreur maximale.

30. Comparer avec `robot.turn()`

Faire cinq essais avec :

```
robot.turn(90)
```

Puis cinq essais avec :

```
tourner_precis(90)
```

Comparer.

Méthode	Erreur moyenne	Erreur maximale
`robot.turn(90)`		
Gyroscope		

31. Quelle méthode est meilleure ?

Il n'existe pas forcément une réponse universelle.

`DriveBase` peut être :

```
simple
```

```
rapide
```

```
suffisamment précis
```

Le gyroscope permet :

```
une mesure indépendante des roues
```

```
une correction en temps réel
```

```
une vérification de l'orientation
```

Le choix dépend du besoin.

32. Le gyroscope peut aussi maintenir une direction

Imaginons que le robot doive avancer parfaitement droit.

Nous voulons :

```
angle cible = 0°
```

Mais si le robot dérive :

```
angle = 5°
```

nous pouvons corriger.

33. Maintien de cap simple

```
gyro.reset_angle(0)

angle_cible = 0

kp = 2

while True:

    angle = gyro.angle()

    erreur = (
        angle_cible
        - angle
    )

    correction = erreur * kp

    robot.drive(
        200,
        correction
    )
```

Le robot essaie de maintenir l'angle à zéro.

34. Exemple

Si :

```
angle = 10°
```

et :

```
cible = 0°
```

alors :

```
erreur = moins 10
```

La correction devient négative.

Le robot tourne légèrement pour revenir vers la direction initiale.

35. Avancer sur une distance en maintenant le cap

```
def avancer_droit(distance):

    gyro.reset_angle(0)
    robot.reset()

    kp = 2

    while robot.distance() < distance:

        erreur = (
            0
            - gyro.angle()
        )

        correction = erreur * kp

        robot.drive(
            200,
            correction
        )

    robot.stop()
```

Puis :

```
avancer_droit(1000)
```

36. Exercice pratique 5

Tracer une ligne droite au sol.

Faire parcourir :

```
2 mètres
```

avec :

```
robot.straight(2000)
```

Puis avec :

```
avancer_droit(2000)
```

Comparer la déviation finale.

37. Pourquoi un robot dévie t-il ?

Même avec deux moteurs identiques, plusieurs facteurs peuvent provoquer une dérive.

```
frottements différents

sol irrégulier

roues légèrement différentes

poids mal réparti

petite erreur mécanique
```

Le gyroscope permet au robot de détecter cette dérive.

38. Limite du gyroscope

Le gyroscope n'est pas parfait.

Il peut présenter :

```
du bruit

une petite dérive

un décalage initial
```

Il est donc important de :

```
laisser le robot immobile lors de l'initialisation

remettre l'angle à zéro au bon moment

tester les valeurs avant la mission
```

39. Test de stabilité

Laisser le robot complètement immobile.

Exécuter :

```
while True:

    print(gyro.angle())

    wait(500)
```

Observer pendant une minute.

Si l'angle évolue alors que le robot ne bouge pas, nous observons une dérive.

40. Pourquoi cette dérive est-elle gênante ?

Supposons que le robot soit immobile mais que le gyroscope indique progressivement :

```
0°
1°
2°
3°
```

Un programme de maintien de direction pourrait croire que le robot tourne.

Il essaierait alors de corriger une rotation qui n'existe pas réellement.

41. Calibration pratique

Avant le départ :

```
placer le robot immobile
```

Puis :

```
gyro.reset_angle(0)
```

Éviter de déplacer le robot pendant cette préparation.

42. Combiner gyroscope et machine à états

Nous pouvons remplacer un état de rotation basé sur :

```
robot.angle()
```

par une rotation basée sur le gyroscope.

Par exemple :

```
elif etat == "TOURNER":
    robot.drive(0, 80)
    if gyro.angle() >= 90:
        robot.stop()
        etat = "AVANCER"
```

43. Attention au reset

Lorsque nous entrons dans `TOURNER`, nous devons faire :

```
gyro.reset_angle(0)
```

une seule fois.

Pas à chaque cycle de l'état.

44. Erreur classique

Incorrect :

```
elif etat == "TOURNER":
    gyro.reset_angle(0)
    robot.drive(0, 80)
    if gyro.angle() >= 90:
        etat = "AVANCER"
```

Pourquoi ?

À chaque boucle, l'angle est remis à zéro.

Il n'atteindra donc jamais 90 degrés.

45. Bonne méthode

Faire le reset lors de la transition vers l'état.

```
if obstacle:
    gyro.reset_angle(0)
    etat = "TOURNER"
```

Puis :

```
elif etat == "TOURNER":
    robot.drive(0, 80)
    if gyro.angle() >= 90:
        robot.stop()
        etat = "AVANCER"
```

46. Cette idée est générale

Lors d'une transition, nous pouvons préparer le nouvel état.

Par exemple :

```
remettre une distance à zéro

remettre un angle à zéro

initialiser un compteur

mémoriser une valeur
```

Puis l'état utilise cette référence pendant plusieurs cycles.

47. Exercice pratique 6

Créer :

```
AVANCER

↓

obstacle

↓

TOURNER_90

↓

AVANCER
```

Utiliser le gyroscope pour mesurer la rotation.

48. Ajouter plusieurs angles

Nous pouvons créer plusieurs états :

```
TOURNER_DROITE
```

et :

```
TOURNER_GAUCHE
```

Mais nous pouvons aussi utiliser :

```
angle_cible = 90
```

Puis l'état `TOURNER` utilise cette variable.

49. Exemple

Avant la transition :

```
angle_cible = -90

gyro.reset_angle(0)

etat = "TOURNER"
```

Puis :

```
elif etat == "TOURNER":

    erreur = (
        angle_cible
        - gyro.angle()
    )

    correction = erreur * kp_angle

    robot.drive(
        0,
        correction
    )

    if abs(erreur) < 2:

        robot.stop()

        etat = "AVANCER"
```

Le même état peut maintenant gérer différents angles.

50. Une fonction pour préparer la rotation

```
def preparer_rotation(angle):

    gyro.reset_angle(0)

    return angle
```

Puis :

```
angle_cible = preparer_rotation(90)

etat = "TOURNER"
```

Cette fonction est optionnelle mais montre comment organiser le programme.

51. Gyroscope et suivi de ligne

Le capteur de couleur reste plus adapté pour suivre une ligne.

Mais le gyroscope peut fournir une information supplémentaire.

Par exemple :

```
Le robot suit une ligne.

Il connaît aussi approximativement son orientation.
```

Cela peut servir pour :

```
détecter un virage

revenir à une orientation connue

stabiliser une trajectoire après un obstacle
```

52. Exemple après évitement

Le robot évite un obstacle puis doit reprendre la même direction générale.

Nous pouvons mémoriser :

```
angle_depart = gyro.angle()
```

Puis, après l'évitement, essayer de revenir vers cette orientation.

Cette approche devient plus avancée mais montre l'intérêt de conserver des références.

53. Angle absolu et angle relatif

Il faut distinguer deux idées.

Angle relatif

```
gyro.reset_angle(0)
```

Puis mesurer un nouveau virage.

Exemple :

```
Tourne de 90° par rapport à maintenant.
```

Référence générale

On évite de faire un reset après chaque action et on utilise une orientation commune pendant toute la mission.

Exemple :

```
0° = direction de départ

90° = direction droite

180° = direction opposée
```

Cette deuxième méthode permet de raisonner sur l'orientation globale du robot.

54. Exemple de navigation

Si nous définissons :

```
Nord = 0°

Est = 90°

Sud = 180°

Ouest = moins 90°
```

nous pouvons demander au robot de rejoindre une orientation donnée.

C'est une première idée de navigation.

55. Attention

Le gyroscope EV3 n'est pas une boussole.

Il ne connaît pas réellement :

```
le nord
```

Nous choisissons simplement :

```
la direction initiale = 0°
```

Toutes les autres orientations sont mesurées par rapport à cette référence.

56. Défi pratique 7 : carré avec gyroscope

Créer un carré de :

```
40 cm de côté
```

Après chaque côté :

```
tourner précisément de 90°
```

Programme possible :

```
for i in range(4):
    robot.straight(400)
    tourner_precis(90)
```

Attention : comme `tourner_precis()` remet le gyroscope à zéro à chaque fois, chaque rotation est relative.

57. Comparaison avec un carré classique

Version classique :

```
for i in range(4):
    robot.straight(400)
    robot.turn(90)
```

Comparer les deux trajectoires.

Noter l'écart final entre :

```
point de départ
```

et :

```
point d'arrivée
```

58. Défi pratique 8 : triangle

Créer un triangle équilatéral.

L'angle extérieur nécessaire est :

```
120°
```

Programme :

```
for i in range(3):
    robot.straight(500)
    tourner_precis(120)
```

59. Défi pratique 9 : maintien de direction

Faire avancer le robot sur :

```
3 mètres
```

en maintenant :

```
0°
```

Tester plusieurs valeurs de :

```
kp
```

Noter :

kp	Déviation finale	Oscillation
0,5		
1		
2		
4		

60. Un `kp` trop faible

Si `kp` est trop faible :

```
le robot corrige trop lentement
```

Il peut conserver une dérive importante.

61. Un `kp` trop élevé

Si `kp` est trop élevé :

```
le robot surcorrige
```

Il peut zigzaguer même lorsqu'il doit avancer droit.

Nous retrouvons exactement les notions du suivi de ligne proportionnel.

62. Le même principe dans différents problèmes

Suivi de ligne :

```
erreur = cible_lumiere - valeur
```

Maintien d'angle :

```
erreur = angle_cible - angle
```

Distance à un obstacle :

```
erreur = distance_cible - distance
```

Dans tous les cas :

```
MESURE
↓
COMPARAISON AVEC UNE CIBLE
↓
ERREUR
↓
CORRECTION
```

C'est un principe fondamental de la robotique.

63. Combiner plusieurs régulations

Un robot plus avancé pourrait devoir :

```
maintenir une distance
```

tout en :

```
maintenant une direction
```

ou :

```
suivre une ligne
```

Cela devient plus complexe car plusieurs contrôleurs peuvent vouloir modifier les moteurs en même temps.

Il faut alors définir clairement quel contrôleur est responsable de chaque action.

64. Défi pratique 10 : rotation proportionnelle

Créer une fonction :

```
def rotation_p(angle_cible, kp):
```

qui utilise :

```
erreur

correction proportionnelle

tolérance

limitation
```

65. Solution possible

```
def rotation_p(angle_cible, kp):

    gyro.reset_angle(0)

    while True:

        erreur = (
            angle_cible
            - gyro.angle()
        )

        if abs(erreur) < 2:
            break

        correction = (
            erreur
            * kp
        )

        correction = limiter(
            correction,
            -150,
            150
        )

        robot.drive(
            0,
            correction
        )

    robot.stop()
```

66. Défi final : parcours orienté

Créer un parcours :

```
Avancer 50 cm

Tourner 90°

Avancer 30 cm

Tourner moins 90°

Avancer 50 cm

Demi tour

Retour
```

Contraintes :

1. toutes les rotations doivent utiliser le gyroscope
2. au moins une rotation doit utiliser une correction proportionnelle
3. la trajectoire doit terminer le plus près possible du point de départ
4. tu dois mesurer l'erreur finale

67. Exemple de structure

```
robot.straight(500)

tourner_precis(90)

robot.straight(300)

tourner_precis(-90)

robot.straight(500)

tourner_precis(180)
```

Puis compléter le trajet selon le parcours construit.

68. Ajouter le gyroscope au robot autonome

Notre robot final pourra maintenant utiliser :

```
capteur de couleur
↓
suivi de ligne
ultrason
↓
détection d'obstacle
tactile
↓
sécurité en cas de collision
gyroscope
↓
orientation précise
```

Nous avons donc plusieurs moyens complémentaires d'observer et de contrôler le robot.

69. Exemple de transition précise

Dans une machine à états :

```
if etat == "SUIVRE":

    if obstacle_proche(200):

        robot.stop()

        gyro.reset_angle(0)

        angle_cible = 90

        etat = "TOURNER"
```

Puis :

```
elif etat == "TOURNER":

    erreur = (
        angle_cible
        - gyro.angle()
    )

    correction = (
        erreur
        * kp_angle
    )

    robot.drive(
        0,
        correction
    )

    if abs(erreur) < 2:

        robot.stop()

        etat = "CONTOURNER"
```

La rotation est maintenant contrôlée par le gyroscope.

70. Questions de fin de cours

Question 1

Que mesure un gyroscope ?

Question 2

À quoi sert :

```
gyro.angle()
```

Question 3

À quoi sert :

```
gyro.reset_angle(0)
```

Question 4

Pourquoi le signe de l'angle est il important ?

Question 5

Pourquoi une rotation très rapide peut elle dépasser l'angle demandé ?

Question 6

Comment calculer une erreur angulaire ?

Question 7

Pourquoi une correction proportionnelle est elle intéressante pour une rotation ?

Question 8

Pourquoi utilise t on une tolérance ?

Question 9

Comment le gyroscope peut il aider un robot à avancer droit ?

Question 10

Pourquoi ne faut il pas remettre le gyroscope à zéro à chaque cycle d'un état de rotation ?

Correction

Réponse 1

Il mesure la rotation du robot autour de son axe.

Réponse 2

À obtenir l'angle actuellement mesuré.

Réponse 3

À considérer l'orientation actuelle comme la nouvelle référence zéro.

Réponse 4

Parce qu'il permet de distinguer les deux sens de rotation.

Réponse 5

À cause de l'inertie et du temps de réaction du système.

Réponse 6

Par exemple :

```
erreur = angle_cible - angle_mesure
```

Réponse 7

Parce que le robot peut tourner rapidement lorsqu'il est loin de la cible et ralentir lorsqu'il s'en approche.

Réponse 8

Parce qu'il n'est pas nécessaire d'atteindre exactement une valeur comme 90,000°.

Réponse 9

En mesurant la déviation par rapport à l'angle souhaité et en corrigeant continuellement la direction.

Réponse 10

Parce que la mesure reviendrait constamment à zéro et l'angle cible ne serait jamais atteint.

Difficulté fréquente

Le robot tourne sans fin.

Causes possibles :

```
signe de rotation incorrect

condition incorrecte

gyroscope remis à zéro dans la boucle

capteur sur le mauvais port
```

La première chose à faire est :

```
print(gyro.angle())
```

et observer si la valeur évolue dans le sens attendu.

Difficulté fréquente

Le robot oscille autour de 90 degrés.

Cela peut arriver avec une correction proportionnelle trop forte.

Solutions :

```
réduire kp

ajouter une tolérance

limiter la correction

ralentir près de la cible
```

Cours 24 : Conception du projet final autonome de la première partie avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. analyser un problème robotique complet
2. transformer un cahier des charges en tâches simples
3. choisir les capteurs adaptés
4. définir les états du robot
5. définir les transitions entre les états
6. établir les priorités
7. identifier les fonctions nécessaires
8. identifier les paramètres à calibrer
9. prévoir les situations d'erreur
10. construire une architecture avant de programmer
11. préparer un plan de tests
12. produire le dossier de conception du projet final

Cette leçon est principalement consacrée à :

```
ANALYSER

CONCEVOIR

PLANIFIER

TESTER SUR PAPIER
```

La programmation complète sera réalisée lors du cours suivant.

1. Introduction

Depuis le début du cours, nous avons appris les différentes briques nécessaires à la création d'un robot autonome.

Nous savons maintenant utiliser :

```
variables

calculs

conditions

boucles

fonctions

listes

moteurs

DriveBase

capteur tactile
```

```
capteur ultrason

capteur de couleur

gyroscope

suivi de ligne

contrôle proportionnel

PID

machines à états

priorités

débogage
```

Le projet final va réunir plusieurs de ces notions.

Mais avant de programmer, nous allons apprendre une étape essentielle :

concevoir le système

2. Pourquoi ne pas commencer directement à coder ?

Lorsqu'un projet devient complexe, commencer immédiatement par écrire du code provoque souvent :

```
un programme désorganisé

des fonctions inutiles

des conditions contradictoires

des états manquants

des bugs difficiles à trouver
```

Une meilleure méthode est :

```
COMPRENDRE LE PROBLÈME
↓
DÉCOUPER LE PROBLÈME
↓
CHOISIR UNE STRATÉGIE
↓
CONCEVOIR LE PROGRAMME
↓
PROGRAMMER
↓
TESTER
```

3. Le projet final

Le robot doit réaliser une mission autonome.

Exemple de mission :

```
DÉPART
↓
rejoindre une ligne
↓
suivre la ligne
↓
détecter un obstacle
↓
éviter l'obstacle
↓
retrouver la ligne
↓
continuer
↓
effectuer une rotation précise
↓
continuer le parcours
↓
détecter la zone d'arrivée
↓
STOP
```

Le parcours exact peut être adapté par le professeur.

4. Cahier des charges

Le robot doit fonctionner sans intervention humaine après son démarrage.

Il doit au minimum :

1. se déplacer de manière autonome
2. utiliser au moins deux capteurs différents
3. suivre une ligne
4. détecter un obstacle
5. réagir à cet obstacle
6. utiliser une machine à états
7. utiliser plusieurs fonctions
8. utiliser au moins une boucle
9. utiliser plusieurs conditions
10. posséder un état d'arrêt
11. détecter une arrivée
12. être capable de terminer la mission

5. Contraintes supplémentaires possibles

Le professeur peut également imposer :

```
utilisation du gyroscope

utilisation du tactile comme sécurité

utilisation d'un PID

nombre maximum de collisions

temps maximum

zone interdite

plusieurs obstacles

plusieurs virages
```

6. Première étape : comprendre la mission

Avant de programmer, chaque étudiant doit répondre à :

```
Que doit faire le robot ?
```

Puis :

```
Dans quel ordre ?
```

Puis :

```
Quels événements peuvent modifier ce comportement ?
```

7. Décomposer la mission

Prenons :

```
Démarrer

Suivre la ligne

Éviter obstacle

Retrouver ligne

Continuer

S'arrêter sur rouge
```

Nous pouvons immédiatement identifier plusieurs comportements.

```
DEMARRER

SUIVRE

EVITER
```

```
CHERCHER

STOP
```

Ces comportements pourront devenir des états.

8. Identifier les capteurs

Pour chaque tâche, demander :

De quelle information le robot a t-il besoin ?

Exemple :

Tâche	Information	Capteur
Suivre ligne	Réflexion du sol	Capteur couleur
Détecter obstacle	Distance	Ultrason
Détecter collision	Contact	Tactile
Rotation précise	Angle	Gyroscope
Détecter arrivée	Couleur rouge	Capteur couleur

9. Choisir uniquement les capteurs utiles

Utiliser quatre capteurs uniquement parce qu'ils sont disponibles n'est pas forcément une bonne stratégie.

Chaque capteur doit avoir une fonction claire.

Demander :

```
Pourquoi ai-je besoin de ce capteur ?
```

Si aucune réponse précise n'existe, il n'est peut être pas nécessaire.

10. Identifier les actions

Le robot peut avoir besoin de :

```
avancer

reculer

tourner

suivre une ligne

chercher la ligne

s'arrêter

éviter un obstacle
```

Ces actions pourront devenir des fonctions.

11. Exemple de fonctions

```
def suivre_ligne_cycle():
def obstacle_proche():
def collision():
def zone_arrivee():
def limiter():
def calculer_correction():
```

Le programme principal ne doit pas contenir tous les détails techniques.

12. Penser en niveaux

Un bon programme peut être organisé en plusieurs niveaux.

```
NIVEAU 1

MISSION
NIVEAU 2

ÉTATS
NIVEAU 3

FONCTIONS
NIVEAU 4

CAPTEURS ET MOTEURS
```

13. Exemple

Mission :

```
ÉVITER UN OBSTACLE
```

peut être composée de :

```
RECULER

TOURNER

CONTOURNER

CHERCHER LA LIGNE
```

Chaque comportement peut lui même être composé d'instructions moteurs.

14. Concevoir la machine à états

Une première proposition :

```
DEMARRER
↓
CHERCHER_LIGNE
↓
SUIVRE
↓
RECULER
↓
TOURNER
↓
CONTOURNER
↓
CHERCHER_LIGNE
↓
SUIVRE
↓
STOP
```

15. Représentation graphique

```
[DEMARRER]
     |
     v
[CHERCHER]
     |
     | ligne détectée
     v
[SUIVRE]
     |
     | obstacle
     v
[RECULER]
     |
     v
[TOURNER]
     |
     v
[CONTOURNER]
     |
     v
[CHERCHER]
     |
     v
[SUIVRE]
     |
     | arrivée rouge
     v
[STOP]
```

16. Pourquoi dessiner avant de coder ?

Ce dessin permet de vérifier immédiatement :

```
Tous les états sont-ils présents ?

Chaque état possède t-il une sortie ?

Le robot peut-il revenir vers SUIVRE ?

Le robot peut-il atteindre STOP ?
```

Il est beaucoup plus simple de corriger le diagramme que plusieurs centaines de lignes de code.

17. Tableau des états

Chaque étudiant doit remplir :

État	Action principale	Condition de sortie	État suivant
DEMARRER			
CHERCHER			
SUIVRE			
RECULER			
TOURNER			
CONTOURNER			
STOP			

18. Exemple

État	Action principale	Condition de sortie	État suivant
DEMARRER	Initialisation	Robot prêt	CHERCHER
CHERCHER	Avancer lentement	Ligne détectée	SUIVRE
SUIVRE	Suivi de ligne	Obstacle	RECULER
RECULER	Reculer	150 mm	TOURNER
TOURNER	Rotation 90°	Angle atteint	CONTOURNER
CONTOURNER	Avancer	Distance atteinte	CHERCHER
STOP	Arrêt moteurs	Aucune	Fin

19. Identifier les transitions globales

Certaines transitions peuvent être valables depuis plusieurs états.

Par exemple :

```
ROUGE
↓
STOP
```

Nous pouvons considérer cette règle comme globale.

Même si le robot est dans :

```
SUIVRE

CHERCHER

CONTOURNER
```

une zone d'arrivée peut provoquer l'arrêt.

20. Définir les priorités

Chaque étudiant doit établir une hiérarchie.

Exemple :

```
Priorité 1

STOP d'urgence
Priorité 2

Zone d'arrivée
Priorité 3

Collision
Priorité 4

Obstacle
Priorité 5

Ligne perdue
Priorité 6

Comportement normal
```

21. Pourquoi décider les priorités maintenant ?

Parce qu'un robot peut recevoir plusieurs informations simultanément.

Exemple :

```
rouge détecté
```

```
+
```

```
obstacle détecté
```

Que doit faire le robot ?

Si cette question n'est pas décidée avant la programmation, la logique peut devenir contradictoire.

22. Exercice de priorité

Pour chaque situation, choisir l'action.

Situation 1

```
ligne correcte
```

```
aucun obstacle
```

Action :

```
SUIVRE
```

Situation 2

```
ligne perdue
```

```
obstacle à 10 cm
```

Quelle priorité ?

Réponse logique :

```
OBSTACLE
```

Situation 3

```
rouge détecté
```

```
collision
```

Si le rouge représente la fin de mission :

```
STOP
```

23. Identifier les paramètres

Un robot autonome contient de nombreux paramètres.

Par exemple :

```
vitesse_suivi = 180
distance_obstacle = 200
distance_recul = 150
angle_evitement = 90
kp = 2
seuil_ligne = 45
```

24. Pourquoi regrouper les paramètres ?

Il est plus pratique de placer les valeurs importantes au début du programme.

Par exemple :

```
VITESSE_SUIVI = 180

DISTANCE_OBSTACLE = 200

DISTANCE_RECUL = 150

ANGLE_EVITEMENT = 90

KP_LIGNE = 2
```

Lors des tests, nous pouvons les retrouver rapidement.

25. Valeurs fixes et valeurs calibrées

Certaines valeurs sont choisies.

Exemple :

```
distance obstacle = 200 mm
```

D'autres doivent être mesurées.

Exemple :

```
valeur noir

valeur blanc

seuil de ligne
```

Il faut distinguer :

```
PARAMÈTRE DE STRATÉGIE
```

et :

```
PARAMÈTRE DE CALIBRATION
```

26. Liste des calibrations

Avant le projet final, chaque étudiant doit relever :

```
valeur du noir

valeur du blanc

seuil

distance réelle ultrason

précision du gyroscope

distance réelle parcourue

angle réel pour une commande de 90°
```

27. Fiche de calibration

Élément	Valeur
Noir	
Blanc	
Cible ligne	
Distance obstacle	
Distance recul	
Angle évitement	
`kp` ligne	
`kp` angle	

28. Concevoir les fonctions

Avant de coder, dresser une liste.

Exemple :

```
def zone_arrivee():
```

Responsabilité :

```
détecter le rouge
def obstacle_proche():
```

Responsabilité :

```
détecter un obstacle
def suivre_ligne_cycle():
```

Responsabilité :

```
effectuer une correction de suivi
```

29. Principe d’une fonction claire

Une bonne fonction possède idéalement une responsabilité principale.

Éviter une fonction comme :

```
def tout_faire():
```

qui :

```
lit les capteurs

suit la ligne

évite les obstacles

tourne

gère le son

compte les obstacles
```

Elle deviendrait difficile à comprendre.

30. Exemple de fonctions simples

```
def obstacle_proche():

    return (
        capteur_ultrason.distance()
        < DISTANCE_OBSTACLE
    )
def collision():

    return capteur_tactile.pressed()
def zone_arrivee():

    return (
        capteur_couleur.color()
        == Color.RED
    )
```

31. Fonction de suivi

```
def suivre_ligne_cycle():

    valeur = (
        capteur_couleur.reflection()
    )

    erreur = CIBLE_LIGNE - valeur
    correction = (
        erreur
        * KP_LIGNE
    )

    robot.drive(
        VITESSE_SUIVI,
        correction
    )
```

32. Réfléchir aux erreurs possibles

Avant de lancer le robot, demander :

Qu’est ce qui peut mal se passer ?

Exemples :

```
ligne perdue

obstacle non détecté

collision

virage trop court

virage trop long

robot bloqué

capteur mal calibré
```

33. Prévoir une stratégie de récupération

Un robot autonome ne doit pas seulement fonctionner lorsque tout se passe parfaitement.

Exemple :

```
PROBLÈME

ligne perdue
```

Réaction :

```
CHERCHER
```

Puis :

```
ligne retrouvée

↓

SUIVRE
```

34. Autre exemple

Problème :

```
obstacle non détecté par ultrason
↓
collision tactile
```

Réaction :

```
STOP
↓
RECULER
↓
ÉVITER
```

Le capteur tactile devient une sécurité.

35. Définir ce qui est une erreur fatale

Toutes les erreurs ne doivent pas forcément conduire à `STOP`.

Par exemple :

```
ligne perdue
```

peut-être récupérable.

Mais :

```
trois échecs consécutifs
```

```
robot bloqué
```

```
trop de collisions
```

peuvent entraîner :

```
STOP
```

36. Compteur d’échecs

Nous pouvons prévoir :

```
nombre_echecs = 0
```

Puis :

```
nombre_echecs += 1
```

Si :

```
if nombre_echecs >= 3:
    etat = "STOP"
```

Cela évite qu'un robot reste bloqué indéfiniment.

37. Définir les conditions de réussite

Une mission doit avoir un critère clair.

Par exemple :

```
zone rouge détectée
```

ou :

```
trois tours terminés
```

ou :

```
objet transporté jusqu'à la zone cible
```

Sans critère de réussite, le robot ne sait pas quand terminer.

38. Définir les conditions d'échec

Par exemple :

```
plus de 5 collisions
plus de 3 pertes de ligne
temps maximum dépassé
robot complètement bloqué
```

Ces règles peuvent être utilisées lors de la compétition finale.

39. Concevoir le programme principal

Une bonne architecture pourrait ressembler à :

```
# Imports
# Configuration
# Paramètres
# Fonctions
# Initialisation
# Boucle principale
```

40. Squelette

```
# IMPORTS

# MOTEURS ET CAPTEURS

# PARAMETRES

# FONCTIONS

# INITIALISATION

etat = "DEMARRER"

while True:
    # événements prioritaires

    # machine à états

    if etat == "STOP":
        robot.stop()
        break
```

41. Pourquoi écrire le squelette avant le contenu ?

Cela permet de construire la structure avant les détails.

Chaque étudiant peut ensuite compléter progressivement :

```
fonctions
priorités
états
```

sans perdre l’organisation générale.

42. Développement progressif

Ne jamais essayer de programmer le robot final complet en une seule fois.

Progression conseillée :

```
Étape 1

faire avancer le robot
Étape 2

faire suivre la ligne
Étape 3

détecter obstacle
Étape 4

reculer
Étape 5

tourner
Étape 6

retrouver la ligne
Étape 7

arrêt final
```

43. Tester chaque fonction seule

Avant d'intégrer :

```
suivre_ligne_cycle()
```

dans la machine complète, tester le suivi seul.

Même principe pour :

```
rotation

détection obstacle

détection rouge

recherche de ligne
```

44. Plan de tests

Chaque étudiant doit créer un tableau.

Test	Objectif	Résultat attendu
1	Moteurs	Les deux tournent
2	Couleur	Noir et blanc détectés
3	Ultrason	Obstacle détecté
4	Tactile	Collision détectée
5	Gyroscope	90° mesurés
6	Suivi	Ligne suivie
7	Évitement	Obstacle contourné
8	Retour ligne	Ligne retrouvée
9	Arrivée	Robot s’arrête

45. Ajouter une colonne résultat

Test	Résultat attendu	Résultat réel	Validé
Moteurs	Fonctionnent		
Couleur	Bon contraste		
Ultrason	Bonne distance		
Gyro	Rotation précise		
Suivi	Stable		
Évitement	Réussi		

46. Ne passer à l’étape suivante que si le test fonctionne

Si le suivi de ligne ne fonctionne pas :

```
NE PAS
```

ajouter immédiatement l’évitement.

Sinon deux problèmes différents seront mélangés.

Le développement doit être progressif.

47. Utiliser le débogage du cours précédent

Pendant les tests :

```
print(etat)
print(distance)
print(valeur)
print(erreur)
print(correction)
```

peuvent être temporairement ajoutés.

48. Prévoir les informations de diagnostic

Dans la version de développement :

```
DEBUG = True
```

Puis :

```
if DEBUG:
    print(etat)
```

Pour la version finale :

```
DEBUG = False
```

49. Exemple de fonction de debug

```
def debug(message):

    if DEBUG:
        print(message)
```

Puis :

```
debug("Obstacle détecté")
```

Cela évite d'avoir à supprimer tous les `print()` un par un.

50. Concevoir l'évitement

Il existe plusieurs stratégies possibles.

Stratégie A

```
reculer

tourner à droite

avancer

chercher la ligne
```

Simple mais dépend beaucoup de la position de l’obstacle.

Stratégie B

```
reculer

tourner 90°

avancer le long de l’obstacle

tourner

chercher la ligne
```

Plus complexe mais plus contrôlée.

51. Comparer les stratégies

Stratégie	Avantage	Inconvénient
Simple	Facile à programmer	Peu robuste
Contournement complet	Plus contrôlé	Plus complexe
Recherche directe	Rapide	Peut rater la ligne

Chaque étudiant peut choisir une stratégie différente.

52. Utiliser le gyroscope pour l’évitement

Exemple :

```
obstacle
↓
reculer 150 mm
↓
tourner 90°
↓
avancer 300 mm
↓
tourner moins 90°
↓
chercher la ligne
```

Le gyroscope peut rendre les rotations plus répétables.

53. Penser à la géométrie

Le robot possède une taille réelle.

Un obstacle n’est pas un simple point.

Il faut tenir compte :

```
largeur du robot

longueur du robot

rayon des virages

position du capteur
```

La stratégie doit donc être testée physiquement.

54. Mesurer le robot

Chaque étudiant peut relever :

Mesure	Valeur
Largeur du robot	
Longueur du robot	
Distance capteur couleur au centre	
Distance ultrason à l’avant	
Écart entre roues	

Ces mesures peuvent expliquer certains comportements.

55. Prévoir la vitesse

Une vitesse élevée est intéressante pour obtenir un bon temps.

Mais elle augmente :

```
l’inertie

le risque de perte de ligne

la distance de freinage

les erreurs de rotation
```

Il faut trouver un équilibre.

56. Vitesse selon l’état

La vitesse ne doit pas forcément être identique partout.

Par exemple :

```
VITESSE_SUIVI = 200
VITESSE_RECHERCHE = 80
VITESSE_RECUL = 120
VITESSE_CONTOURNEMENT = 150
```

57. Pourquoi ?

Le robot peut être rapide lorsqu’il connaît bien sa trajectoire.

Mais pendant :

`la recherche de ligne`

il vaut mieux parfois ralentir pour éviter de dépasser la ligne.

58. Concevoir les variables globales

Exemple :

```
etat = "DEMARRER"

nombre_obstacles = 0

nombre_collisions = 0

nombre_pertes = 0
```

Ces variables représentent la mémoire globale de la mission.

59. Variables locales

Dans une fonction :

```
def suivre_ligne_cycle():

    valeur = capteur_couleur.reflection()

    erreur = CIBLE_LIGNE - valeur
```

`valeur` et `erreur` peuvent rester locales.

Elles ne sont utiles que pendant le calcul.

60. Ne pas créer trop de variables globales

Plus le nombre de variables globales augmente, plus le programme devient difficile à comprendre.

Garder global uniquement ce qui doit réellement être partagé entre plusieurs parties.

61. Diagramme de flux

En complément de la machine à états, on peut dessiner :

```
DÉBUT
↓
Initialisation
↓
Lire capteurs
↓
Arrivée ?

Oui → STOP

Non
↓
Collision ?

Oui → RECULER

Non
↓
Exécuter état actuel
↓
Recommencer
```

62. Exercice pratique 1

Chaque étudiant reçoit le cahier des charges.

Sans ordinateur, tu dois produire :

1. la liste des capteurs
2. la liste des états
3. la liste des fonctions
4. le tableau des transitions
5. l'ordre des priorités

Aucun code complet n'est encore autorisé.

63. Exercice pratique 2

Créer le diagramme d'états sur papier.

Chaque état doit posséder :

`une action`

`une entrée`

`une sortie`

Si un état n'a pas de condition de sortie, l'étudiant doit justifier pourquoi.

`STOP` est le seul état qui peut naturellement ne pas avoir de sortie.

64. Exercice pratique 3

Lister les paramètres à régler.

Exemple :

```
Vitesse suivi
Kp
Distance obstacle
Distance recul
Angle rotation
Vitesse recherche
Seuil ligne
```

Puis prévoir comment chaque paramètre sera testé.

65. Exercice pratique 4

Identifier cinq risques.

Exemple :

Risque	Détection	Réaction
Ligne perdue	Réflexion claire longtemps	CHERCHER
Collision	Tactile	RECULER
Obstacle	Ultrason	EVITER
Rotation incorrecte	Gyroscope	Corriger
Trop d'échecs	Compteur	STOP

66. Exercice pratique 5

Créer le squelette Python complet sans implémenter toutes les fonctions.

Exemple :

```
def obstacle_proche():
    pass


def zone_arrivee():
    pass


def suivre_ligne_cycle():
    pass


etat = "DEMARRER"


while True:

    if etat == "DEMARRER":
        pass

    elif etat == "SUIVRE":
        pass

    elif etat == "RECULER":
        pass

    elif etat == "TOURNER":
        pass

    elif etat == "CHERCHER":
        pass

    elif etat == "STOP":
        break
```

67. Pourquoi utiliser `pass` ?

`pass` permet de créer une structure temporairement vide.

Cela permet de vérifier l'architecture avant d'écrire le contenu.

68. Transformer ensuite `pass`

Par exemple :

```
def obstacle_proche():

    return (
        capteur_ultrason.distance()
        < DISTANCE_OBSTACLE
    )
```

Puis les autres fonctions sont complétées une par une.

69. Revue de conception

Avant de programmer le projet final, chaque étudiant doit présenter son architecture.

Tu dois expliquer :

```
Quels capteurs utilises-tu ?

Pourquoi ?

Quels sont les états ?

Quelles sont les priorités ?

Comment évites-tu un obstacle ?

Comment retrouves-tu la ligne ?

Comment détectes-tu l'arrivée ?

Que se passe t-il en cas d'échec ?
```

70. Le professeur peut jouer le rôle du robot

Une activité utile consiste à lire le diagramme avec les étudiants.

Exemple :

```
Tu es dans SUIVRE.

Je place un obstacle devant toi.

Que fais-tu ?
```

Réponse :

```
RECULER
```

Puis :

```
Le recul est terminé.
```

Réponse :

```
TOURNER
```

Cette simulation permet de détecter des transitions manquantes.

71. Cas inattendus

Tester mentalement :

```
Que se passe t-il si le rouge est détecté pendant RECULER ?
Que se passe t-il si le tactile est appuyé pendant CHERCHER ?
Que se passe t-il si la ligne n'est jamais retrouvée ?
```

Ces questions améliorent fortement la conception.

72. Prévoir un timeout logique

Même sans gérer précisément le temps, on peut utiliser des compteurs.

Par exemple, en état `CHERCHER` :

```
cycles_recherche += 1
```

Puis :

```
if cycles_recherche > 500:
    etat = "STOP"
```

Cela empêche une recherche infinie.

73. Réinitialiser le compteur

Lorsque nous entrons dans `CHERCHER` :

```
cycles_recherche = 0
```

Puis il augmente uniquement dans cet état.

74. Attention au reset dans la boucle

Comme avec le gyroscope, ne pas écrire :

```
elif etat == "CHERCHER":
    cycles_recherche = 0
```

à chaque cycle.

Sinon il ne dépassera jamais zéro.

La réinitialisation doit être faite lors de l'entrée dans l'état.

75. Notion d’action d’entrée

Lorsqu’on entre dans un état, on peut effectuer une préparation.

Par exemple :

```
ENTRÉE DANS TOURNER
↓
gyro.reset_angle(0)
ENTRÉE DANS RECULER
↓
robot.reset()
ENTRÉE DANS CHERCHER
↓
cycles_recherche = 0
```

Cette idée rend les machines à états plus robustes.

76. Concevoir les transitions proprement

Exemple :

```
if obstacle_proche():

    robot.stop()

    robot.reset()

    etat = "RECULER"
```

Le reset fait partie de la préparation du prochain état.

77. Défi de conception

Le professeur ajoute une nouvelle contrainte :

```
Après deux obstacles, le robot doit réduire sa vitesse de 25 %.
```

Les étudiants doivent modifier uniquement leur conception.

Ils doivent indiquer :

```
quelle variable ajouter

où incrémenter le compteur

où modifier la vitesse
```

78. Autre contrainte

```
Après une collision, le robot doit produire deux sons.
```

Quelle fonction pourrait être créée ?

Par exemple :

```
def signal_collision():

    for i in range(2):
        ev3.speaker.beep()
```

79. Autre contrainte

```
Après trois collisions, mission échouée.
```

Nous pouvons prévoir :

```
if nombre_collisions >= 3:

    etat = "STOP"
```

L'architecture doit pouvoir évoluer sans être entièrement réécrite.

80. Critères d'une bonne conception

Une bonne architecture doit être :

```
lisible

modulaire

testable

modifiable

robuste
```

81. Lisible

Un autre étudiant doit pouvoir comprendre approximativement le fonctionnement sans lire chaque ligne.

Des noms comme :

```
obstacle_proche()
```

sont préférables à :

```
test1()
```

82. Modulaire

Les différentes parties sont séparées.

Exemple :

```
détection obstacle

suivi ligne

rotation

recherche
```

83. Testable

Chaque fonction peut être testée indépendamment.

84. Modifiable

Changer :

```
distance obstacle
```

ne doit pas obliger à modifier dix endroits différents.

D'où l'intérêt de :

```
DISTANCE_OBSTACLE = 200
```

85. Robuste

Le robot doit gérer plusieurs situations inattendues.

Pas uniquement le parcours parfait.

86. Dossier de conception à rendre

Chaque étudiant doit produire :

Partie 1

Description de la mission.

Partie 2

Liste des capteurs.

Partie 3

Liste des états.

Partie 4

Diagramme des états.

Partie 5

Tableau des transitions.

Partie 6

Ordre des priorités.

Partie 7

Liste des fonctions.

Partie 8

Liste des paramètres.

Partie 9

Plan de tests.

Partie 10

Risques et stratégies de récupération.

87. Grille de validation de la conception

Élément	Validé
Mission comprise	
Capteurs choisis	
États définis	
Transitions définies	
Priorités définies	
Fonctions définies	
Paramètres identifiés	
Stratégies d’erreur prévues	
Plan de tests créé	
Squelette Python créé	

88. Exemple de conception finale

États

DEMARRER

CHERCHER

SUIVRE

RECULER

TOURNER

CONTOURNER

STOP

Capteurs

Couleur

Ultrason

Tactile

Gyroscope

Priorités

1 STOP urgent

2 arrivée

3 collision

4 obstacle

5 ligne perdue

6 comportement normal

89. Paramètres

```
VITESSE_SUIVI = 180
VITESSE_RECHERCHE = 80
VITESSE_RECUL = 120
DISTANCE_OBSTACLE = 200
DISTANCE_RECUL = 150
ANGLE_EVITEMENT = 90
KP_LIGNE = 2
KP_ANGLE = 2
```

90. Fonctions

```
def obstacle_proche():
def collision():
def zone_arrivee():
def suivre_ligne_cycle():
def limiter():
def diagnostic():
```

91. Structure du programme

```
IMPORTS
↓
INITIALISATION DU MATÉRIEL
↓
PARAMÈTRES
↓
FONCTIONS
↓
CALIBRATION
↓
INITIALISATION DE LA MISSION
↓
BOUCLE PRINCIPALE
↓
PRIORITÉS
↓
MACHINE À ÉTATS
↓
STOP
```

92. Questions de fin de cours

Question 1

Pourquoi faut il concevoir avant de programmer ?

Question 2

Comment déterminer les états ?

Question 3

Quelle différence existe entre une action et une transition ?

Question 4

Pourquoi définir des priorités ?

Question 5

Pourquoi regrouper les paramètres ?

Question 6

Pourquoi tester les fonctions séparément ?

Question 7

Qu’est ce qu’une stratégie de récupération ?

Question 8

Pourquoi prévoir une condition d’échec ?

Question 9

Pourquoi utiliser des noms de fonctions explicites ?

Question 10

Pourquoi le programme final doit il être développé progressivement ?

Correction

Réponse 1

Pour organiser le problème avant d'ajouter la complexité du code.

Réponse 2

En identifiant les différentes situations dans lesquelles le robot peut se trouver.

Réponse 3

Une action décrit ce que fait le robot.

Une transition décrit quand il change d'état.

Réponse 4

Pour savoir quel événement doit être traité lorsqu'il en existe plusieurs simultanément.

Réponse 5

Pour pouvoir modifier facilement le comportement sans rechercher les valeurs partout dans le code.

Réponse 6

Pour isoler les problèmes et faciliter le débogage.

Réponse 7

Un comportement permettant au robot de revenir à une situation normale après un problème.

Réponse 8

Pour éviter que le robot reste bloqué indéfiniment dans une situation impossible.

Réponse 9

Pour rendre le programme compréhensible.

Réponse 10

Parce qu'il est beaucoup plus facile de valider chaque élément avant de les combiner.

Difficulté fréquente

Tu commences à coder une solution très spécifique immédiatement.

Puis lorsqu'une règle change, tout le programme doit être réécrit.

Encourager :

```
paramètres
```

```
fonctions
```

```
états
```

pour rendre le système adaptable.

Difficulté fréquente

Oublier les situations où quelque chose ne fonctionne pas.

Demander systématiquement :

Et si cela échoue ?

Par exemple :

```
Et si la ligne n'est pas retrouvée ?
```

```
Et si l'obstacle est touché ?
```

```
Et si le gyroscope donne un angle incorrect ?
```

```
Et si le robot reste bloqué ?
```

Cours 25 : Projet final de la première partie, essais et compétition avec LEGO EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. transformer une conception en programme complet
2. intégrer plusieurs fonctions dans un même projet
3. utiliser une machine à états complète
4. utiliser plusieurs capteurs simultanément
5. réaliser des tests unitaires
6. réaliser des tests d'intégration
7. calibrer le robot
8. diagnostiquer les erreurs
9. optimiser les paramètres
10. mesurer les performances
11. expliquer ses choix techniques
12. présenter un robot autonome fonctionnel

Matériel nécessaire

1. 1 LEGO Education Mindstorms EV3
2. 2 grands moteurs EV3
3. 1 capteur de couleur EV3
4. 1 capteur ultrason EV3
5. 1 capteur tactile EV3
6. 1 gyroscope EV3
7. 2 roues identiques
8. 1 ordinateur avec Visual Studio Code
9. MicroPython installé sur l'EV3
10. la piste finale
11. plusieurs obstacles
12. une zone rouge d'arrivée
13. chronomètre
14. feuilles de résultats

1. Introduction

Nous arrivons à la dernière leçon.

Depuis le premier cours, nous sommes passés de :

```
ev3.speaker.beep()
```

à des robots capables de :

```
mesurer leur environnement

prendre des décisions

suivre une ligne

éviter des obstacles

contrôler leur orientation

gérer plusieurs capteurs

changer de comportement

récupérer après certaines erreurs
```

L'objectif final est maintenant de réunir toutes ces compétences dans un seul projet.

2. La mission finale

Le robot doit réaliser automatiquement un parcours contenant plusieurs difficultés.

Exemple :

```
DÉPART
↓
chercher la ligne
↓
suivre la ligne
↓
détecter un obstacle
↓
éviter l'obstacle
↓
retrouver la ligne
↓
continuer
↓
effectuer une rotation précise
↓
continuer le suivi
↓
détecter la zone rouge
↓
STOP
```

Une fois le bouton de démarrage activé, aucune intervention humaine n'est autorisée.

3. Compétences mobilisées

Le projet final peut utiliser :

```
variables

calculs

conditions

boucles

fonctions

listes

DriveBase

capteur tactile

capteur ultrason

capteur de couleur

gyroscope

contrôle proportionnel

PID

machine à états

priorités

débogage
```

Il n’est cependant pas obligatoire d’utiliser chaque notion si elle n’apporte rien au projet.

Le programme doit rester pertinent.

4. Première règle

Ne pas commencer par exécuter immédiatement le programme complet.

Nous allons utiliser une démarche progressive.

```
VÉRIFIER LE MATÉRIEL
↓
TESTER LES CAPTEURS
↓
TESTER LES ACTIONS
↓
TESTER LES FONCTIONS
↓
TESTER LES ÉTATS
↓
COMBINER
↓
TESTER LE PARCOURS COMPLET
```

5. Vérification physique

Avant de programmer, vérifier :

```
roues correctement fixées

moteurs correctement fixés

capteurs stables

câbles correctement connectés

aucun câble dans les roues

robot bien équilibré
```

Un problème mécanique peut rendre impossible le réglage du programme.

6. Vérifier les ports

Exemple de configuration :

```
B = moteur gauche

C = moteur droit

S1 = tactile

S2 = gyroscope

S3 = couleur

S4 = ultrason
```

Le programme doit utiliser exactement la même configuration.

7. Initialisation du projet

Exemple :

```
from pybricks.hubs import EV3Brick

from pybricks.ev3devices import (
    Motor,
    TouchSensor,
    ColorSensor,
    UltrasonicSensor,
    GyroSensor
)

from pybricks.parameters import (
    Port,
    Color
)

from pybricks.robotics import DriveBase

from pybricks.tools import wait
```

Puis :

```
ev3 = EV3Brick()

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_tactile = TouchSensor(Port.S1)
gyro = GyroSensor(Port.S2)
capteur_couleur = ColorSensor(Port.S3)
capteur_ultrason = UltrasonicSensor(Port.S4)
```

Et enfin :

```
robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)
```

8. Regrouper les paramètres

Exemple :

```
VITESSE_SUIVI = 180
VITESSE_RECHERCHE = 80
VITESSE_RECUL = 120
VITESSE_CONTOURNEMENT = 140

DISTANCE_OBSTACLE = 220
DISTANCE_RECUL = 150
DISTANCE_CONTOURNEMENT = 300

ANGLE_EVITEMENT = 90

KP_LIGNE = 2.0
KP_ANGLE = 2.0

LIMITE_CORRECTION = 180
```

Toutes les valeurs importantes sont regroupées.

9. Calibration du capteur de couleur

Chaque étudiant doit mesurer sa propre piste.

Exemple :

```
VALEUR_NOIR = 15
VALEUR_BLANC = 75
```

Puis :

```
CIBLE_LIGNE = (
    VALEUR_NOIR
    + VALEUR_BLANC
) / 2
```

Ici :

```
CIBLE_LIGNE = 45
```

10. Vérifier le contraste

Si :

```
Noir = 15
```

```
Blanc = 75
```

le contraste est important.

Si :

```
Noir = 40
```

```
Blanc = 48
```

le suivi sera beaucoup plus difficile.

Il faut alors vérifier :

```
hauteur du capteur
```

```
éclairage
```

```
surface
```

```
position du capteur
```

11. Fonction `limiter()`

Nous réutilisons :

```
def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur
```

Cette fonction permettra de sécuriser certaines corrections.

12. Fonction `obstacle_proche()`

```
def obstacle_proche():

    return (
        capteur_ultrason.distance()
        < DISTANCE_OBSTACLE
    )
```

13. Fonction `collision()`

```
def collision():

    return capteur_tactile.pressed()
```

14. Fonction `zone_arrivee()`

```
def zone_arrivee():

    return (
        capteur_couleur.color()
        == Color.RED
    )
```

15. Fonction de suivi proportionnel

```
def suivre_ligne_cycle():

    valeur = (
        capteur_couleur.reflection()
    )

    erreur = (
        CIBLE_LIGNE
        - valeur
    )

    correction = (
        erreur
        * KP_LIGNE
    )

    correction = limiter(
        correction,
        -LIMITE_CORRECTION,
        LIMITE_CORRECTION
    )

    robot.drive(
        VITESSE_SUIVI,
        correction
    )

    return valeur
```

16. Pourquoi retourner `valeur` ?

La fonction réalise le suivi.

Mais le programme principal peut également utiliser la valeur pour détecter une perte de ligne.

```
valeur = suivre_ligne_cycle()
```

Puis :

```
if valeur > LIMITE_PERTE:
```

17. Détection d’une perte de ligne

Définissons :

```
LIMITE_PERTE = 70
```

Puis :

```
compteur_perte = 0
```

Dans l’état `SUIVRE` :

```
if valeur > LIMITE_PERTE:

    compteur_perte += 1

else:

    compteur_perte = 0
```

18. Transition vers CHERCHER

```
if compteur_perte > 20:

    robot.stop()

    compteur_perte = 0

    etat = "CHERCHER"
```

Le robot ne considère donc pas une seule mesure incorrecte comme une véritable perte.

19. Rotation avec gyroscope

Créons une fonction de correction pour la rotation.

```
def correction_angle(angle_cible):

    erreur = (
        angle_cible
        - gyro.angle()
    )

    correction = (
        erreur
        * KP_ANGLE
    )

    correction = limiter(
        correction,
        -150,
        150
    )

    return erreur, correction
```

20. Pourquoi retourner deux valeurs ?

Nous avons besoin :

```
de l'erreur
```

pour savoir si la rotation est terminée.

Et :

```
de la correction
```

pour commander le robot.

Nous pouvons écrire :

```
erreur_angle, correction = correction_angle(
    angle_cible
)
```

21. États du robot

Nous utiliserons par exemple :

```
DEMARRER

CHERCHER

SUIVRE

RECULER

TOURNER_1

CONTOURNER

TOURNER_2

RECHERCHER_LIGNE

STOP
```

Le projet peut avoir plus ou moins d'états selon la stratégie choisie.

22. Initialisation des variables

```
etat = "DEMARRER"

nombre_obstacles = 0
nombre_collisions = 0
compteur_perte = 0

angle_cible = 0

DEBUG = True
```

23. Fonction de débogage

```
def debug(message):

    if DEBUG:
        print(message)
```

Nous pouvons utiliser :

```
debug("Passage vers RECULER")
```

24. État DEMARRER

L'état de démarrage peut :

```
arrêter les moteurs

mettre le gyroscope à zéro

produire un son

préparer les compteurs
```

Exemple :

```
if etat == "DEMARRER":

    robot.stop()

    gyro.reset_angle(0)

    ev3.speaker.beep()

    etat = "CHERCHER"
```

25. État CHERCHER

Le robot cherche la ligne.

```
elif etat == "CHERCHER":

    robot.drive(
        VITESSE_RECHERCHE,
        0
    )

    if (
        capteur_couleur.reflection()
        < CIBLE_LIGNE
    ):

        robot.stop()

        etat = "SUIVRE"
```

26. État SUIVRE

```
elif etat == "SUIVRE":

    valeur = suivre_ligne_cycle()

    if obstacle_proche():

        robot.stop()

        nombre_obstacles += 1

        robot.reset()

        etat = "RECULER"
```

Nous pouvons ensuite ajouter la gestion de ligne perdue.

27. Ligne perdue dans SUIVRE

```
if valeur > LIMITE_PERTE:

    compteur_perte += 1

else:

    compteur_perte = 0


if compteur_perte > 20:

    robot.stop()

    compteur_perte = 0

    etat = "RECHERCHER_LIGNE"
```

28. Attention à l'ordre

Si un obstacle et une perte de ligne sont détectés en même temps, l'obstacle doit normalement être prioritaire.

Nous pouvons donc écrire :

```
if obstacle_proche():

    # évitement

elif compteur_perte > 20:

    # recherche de ligne
```

29. État RECULER

```
elif etat == "RECULER":

    robot.drive(
        -VITESSE_RECUL,
        0
    )

    if (
        robot.distance()
        <= -DISTANCE_RECUL
    ):

        robot.stop()

        gyro.reset_angle(0)

        angle_cible = ANGLE_EVITEMENT

        etat = "TOURNER_1"
```

Le reset du gyroscope se fait avant d'entrer dans l'état de rotation.

30. État TOURNER_1

```
elif etat == "TOURNER_1":

    erreur_angle, correction = (
        correction_angle(
            angle_cible
        )
    )

    robot.drive(
        0,
        correction
    )

    if abs(erreur_angle) < 2:

        robot.stop()

        robot.reset()

        etat = "CONTOURNER"
```

31. État CONTOURNER

```
elif etat == "CONTOURNER":
    robot.drive(
        VITESSE_CONTOURNEMENT,
        0
    )

    if (
        robot.distance()
        >= DISTANCE_CONTOURNEMENT
    ):

        robot.stop()

        gyro.reset_angle(0)

        angle_cible = (
            -ANGLE_EVITEMENT
        )

        etat = "TOURNER_2"
```

32. État TOURNER_2

```
elif etat == "TOURNER_2":
    erreur_angle, correction = (
        correction_angle(
            angle_cible
        )
    )

    robot.drive(
        0,
        correction
    )

    if abs(erreur_angle) < 2:
        robot.stop()

        etat = "RECHERCHER_LIGNE"
```

33. État RECHERCHER_LIGNE

```
elif etat == "RECHERCHER_LIGNE":
    robot.drive(
        VITESSE_RECHERCHE,
        -40
    )

    if (
        capteur_couleur.reflection()
        < CIBLE_LIGNE
    ):

        robot.stop()

        compteur_perte = 0

        etat = "SUIVRE"
```

34. État STOP

```
elif etat == "STOP":

    robot.stop()
    ev3.speaker.beep()
    break
```

35. Ajouter les priorités globales

Avant la machine à états, nous pouvons vérifier :

```
if zone_arrivee():
    etat = "STOP"
```

Puis :

```
elif collision():

    nombre_collisions += 1
    robot.stop()
    robot.reset()
    etat = "RECULER"
```

36. Attention à la collision dans certains états

Si le robot est déjà en train de reculer, le tactile pourrait encore rester appuyé pendant un instant.

Nous ne voulons pas réinitialiser continuellement l'état `RECULER`.

Nous pouvons donc ajouter :

```
elif (
    collision()
    and etat != "RECULER"
    and etat != "STOP"
):
```

37. Exemple

```
if zone_arrivee():
    etat = "STOP"

elif (
    collision()
    and etat != "RECULER"
    and etat != "STOP"
):

    nombre_collisions += 1

    robot.stop()
    robot.reset()

    etat = "RECULER"
```

38. Limiter le nombre de collisions

Nous pouvons décider :

```
MAX_COLLISIONS = 3
```

Puis :

```
if nombre_collisions >= MAX_COLLISIONS:

    debug("Trop de collisions")

    etat = "STOP"
```

Le projet possède maintenant une condition d'échec.

39. Version globale du programme

Voici une structure complète que tu peux utiliser comme référence.

```
from pybricks.hubs import EV3Brick

from pybricks.ev3devices import (
    Motor,
    TouchSensor,
    ColorSensor,
    UltrasonicSensor,
    GyroSensor
)

from pybricks.parameters import (
    Port,
    Color
)

from pybricks.robotics import DriveBase


ev3 = EV3Brick()

moteur_gauche = Motor(Port.B)
moteur_droit = Motor(Port.C)

capteur_tactile = TouchSensor(Port.S1)
gyro = GyroSensor(Port.S2)
capteur_couleur = ColorSensor(Port.S3)
capteur_ultrason = UltrasonicSensor(Port.S4)


robot = DriveBase(
    moteur_gauche,
    moteur_droit,
    wheel_diameter=56,
    axle_track=114
)


VITESSE_SUIVI = 180
VITESSE_RECHERCHE = 80
VITESSE_RECUL = 120
VITESSE_CONTOURNEMENT = 140
```

```
DISTANCE_OBSTACLE = 220
DISTANCE_RECUL = 150
DISTANCE_CONTOURNEMENT = 300

ANGLE_EVITEMENT = 90

VALEUR_NOIR = 15
VALEUR_BLANC = 75

CIBLE_LIGNE = (
    VALEUR_NOIR
    + VALEUR_BLANC
) / 2

LIMITE_PERTE = 70

KP_LIGNE = 2.0
KP_ANGLE = 2.0

LIMITE_CORRECTION = 180

MAX_COLLISIONS = 3

DEBUG = True


def debug(message):

    if DEBUG:
        print(message)


def limiter(valeur, minimum, maximum):

    if valeur > maximum:
        return maximum

    if valeur < minimum:
        return minimum

    return valeur


def obstacle_proche():

    return (
        capteur_ultrason.distance()
        < DISTANCE_OBSTACLE
    )


def collision():

    return capteur_tactile.pressed()


def zone_arrivee():

    return (
        capteur_couleur.color()
        == Color.RED
    )


def suivre_ligne_cycle():
```

```
    valeur = (
        capteur_couleur.reflection()
    )

    erreur = (
        CIBLE_LIGNE
        - valeur
    )

    correction = (
        erreur
        * KP_LIGNE
    )

    correction = limiter(
        correction,
        -LIMITE_CORRECTION,
        LIMITE_CORRECTION
    )

    robot.drive(
        VITESSE_SUIVI,
        correction
    )

    return valeur


def correction_angle(angle_cible):

    erreur = (
        angle_cible
        - gyro.angle()
    )

    correction = (
        erreur
        * KP_ANGLE
    )

    correction = limiter(
        correction,
        -150,
        150
    )

    return erreur, correction


etat = "DEMARRER"

nombre_obstacles = 0
nombre_collisions = 0
compteur_perte = 0

angle_cible = 0


while True:

    # PRIORITÉS GLOBALES

    if zone_arrivee():

        debug("Arrivée")

        etat = "STOP"
```

```
elif (
    collision()
    and etat != "RECULER"
    and etat != "STOP"
):

    debug("Collision")

    nombre_collisions += 1

    robot.stop()
    robot.reset()

    etat = "RECULER"


if (
    nombre_collisions
    >= MAX_COLLISIONS
):

    debug("Trop de collisions")

    etat = "STOP"


# MACHINE À ÉTATS

if etat == "DEMARRER":

    debug("DEMARRER")

    robot.stop()

    gyro.reset_angle(0)

    ev3.speaker.beep()

    etat = "CHERCHER"


elif etat == "CHERCHER":

    robot.drive(
        VITESSE_RECHERCHE,
        0
    )

    if (
        capteur_couleur.reflection()
        < CIBLE_LIGNE
    ):

        debug("Ligne trouvée")

        robot.stop()

        etat = "SUIVRE"


elif etat == "SUIVRE":

    valeur = suivre_ligne_cycle()

    if obstacle_proche():
```

```
            debug("Obstacle")

            robot.stop()

            nombre_obstacles += 1

            robot.reset()

            etat = "RECULER"

        else:

            if valeur > LIMITE_PERTE:

                compteur_perte += 1

            else:

                compteur_perte = 0


            if compteur_perte > 20:

                debug("Ligne perdue")

                robot.stop()

                compteur_perte = 0

                etat = "RECHERCHER_LIGNE"


    elif etat == "RECULER":

        robot.drive(
            -VITESSE_RECUL,
            0
        )

        if (
            robot.distance()
            <= -DISTANCE_RECUL
        ):

            robot.stop()

            gyro.reset_angle(0)

            angle_cible = (
                ANGLE_EVITEMENT
            )

            etat = "TOURNER_1"


    elif etat == "TOURNER_1":

        erreur_angle, correction = (
            correction_angle(
                angle_cible
            )
        )

        robot.drive(
            0,
            correction
        )
```

```
    if abs(erreur_angle) < 2:

        robot.stop()

        robot.reset()

        etat = "CONTOURNER"


elif etat == "CONTOURNER":

    robot.drive(
        VITESSE_CONTOURNEMENT,
        0
    )

    if (
        robot.distance()
        >= DISTANCE_CONTOURNEMENT
    ):

        robot.stop()

        gyro.reset_angle(0)

        angle_cible = (
            -ANGLE_EVITEMENT
        )

        etat = "TOURNER_2"


elif etat == "TOURNER_2":

    erreur_angle, correction = (
        correction_angle(
            angle_cible
        )
    )

    robot.drive(
        0,
        correction
    )

    if abs(erreur_angle) < 2:

        robot.stop()

        etat = "RECHERCHER_LIGNE"


elif etat == "RECHERCHER_LIGNE":

    robot.drive(
        VITESSE_RECHERCHE,
        -40
    )

    if (
        capteur_couleur.reflection()
        < CIBLE_LIGNE
    ):

        debug("Ligne retrouvée")
```

```
            robot.stop()
            compteur_perte = 0
            etat = "SUIVRE"

    elif etat == "STOP":
        robot.stop()
        debug("STOP")
        ev3.speaker.beep()
        break
```

40. Important

Ce programme est un exemple d'architecture.

Il ne doit pas être considéré comme :

```
une solution universelle
```

Les distances, angles et stratégies doivent être adaptés :

```
au robot
à la piste
aux obstacles
au montage
```

41. Tests unitaires

Avant de lancer ce programme, tester séparément :

```
obstacle_proche()
collision()
zone_arrivee()
suivre_ligne_cycle()
correction_angle()
```

42. Test de `obstacle_proche()`

```
while True:

    print(
        obstacle_proche()
    )
```

Approcher et éloigner un obstacle.

43. Test de `zone_arrivee()`

```
while True:

    print(
        zone_arrivee()
    )
```

Tester la zone rouge.

44. Test du suivi

Ne tester que :

```
while True:

    suivre_ligne_cycle()
```

Le robot doit suivre correctement avant d'ajouter les autres comportements.

45. Test de rotation

```
gyro.reset_angle(0)

while True:

    erreur, correction = (
        correction_angle(90)
    )

    robot.drive(
        0,
        correction
    )

    if abs(erreur) < 2:

        robot.stop()

        break
```

46. Tests d'intégration

Une fois les fonctions validées, tester deux comportements ensemble.

Par exemple :

```
SUIVRE

+

OBSTACLE
```

Puis :

```
OBSTACLE

+

RECULER
```

Puis :

```
RECULER

+

TOURNER
```

Ajouter progressivement les états.

47. Pourquoi cette méthode est-elle importante ?

Si nous testons immédiatement neuf états et quatre capteurs, un problème peut venir de dizaines d'endroits différents.

Si nous ajoutons un élément à la fois, nous savons quelle modification a introduit le problème.

48. Premier essai complet

Le premier essai complet n'a pas pour objectif d'être rapide.

Utiliser des vitesses modérées.

L'objectif est :

```
terminer le parcours
```

sans intervention.

La vitesse sera optimisée ensuite.

49. Fiche premier essai

Élément	Résultat
Ligne trouvée	
Suivi stable	
Obstacle détecté	
Recul correct	
Rotation correcte	
Obstacle contourné	
Ligne retrouvée	
Arrivée détectée	
Mission terminée	

50. Identifier le premier point d’échec

Si la mission échoue, ne corriger qu’un problème à la fois.

Exemple :

```
Le robot suit correctement.

Il détecte l’obstacle.

Il recule correctement.

Il rate la rotation.
```

Nous savons que la priorité est :

```
corriger TOURNER
```

Inutile de modifier le suivi de ligne.

51. Optimisation

Une fois le parcours fiable, nous pouvons optimiser.

Paramètres possibles :

```
VITESSE_SUIVI

KP_LIGNE

DISTANCE_OBSTACLE

DISTANCE_RECUL

ANGLE_EVITEMENT

DISTANCE_CONTOURNEMENT

VITESSE_RECHERCHE
```

52. Règle d’optimisation

Modifier :

```
une seule valeur à la fois
```

Tester.

Mesurer.

Noter.

53. Journal de tests

Essai	Modification	Temps	Collisions	Échec	Observation
1	Référence				
2	Vitesse +20				
3	`kp` modifié				
4	Distance obstacle				
5	Angle évitement				

54. Fiabilité contre vitesse

Le robot A termine en :

```
30 secondes
```

mais réussit seulement :

```
1 fois sur 3
```

Le robot B termine en :

```
35 secondes
```

mais réussit :

```
10 fois sur 10
```

Pour un système autonome réel, le robot B peut être considéré comme meilleur.

55. Mesurer la fiabilité

Effectuer plusieurs essais.

Par exemple :

```
5 parcours
```

Puis :

```
fiabilité = réussites / essais
```

Si :

```
4 réussites sur 5
```

alors :

```
80 %
```

56. Exercice pratique

Chaque étudiant réalise cinq essais.

Essai	Réussi	Temps
1		
2		
3		
4		
5		

Puis calculer :

```
taux de réussite
```

et :

```
temps moyen des essais réussis
```

57. Optimiser le suivi de ligne

Si le robot perd la ligne :

```
diminuer la vitesse
```

```
ajuster Kp
```

```
recalibrer le capteur
```

```
vérifier la hauteur du capteur
```

Ne pas modifier immédiatement toute la machine à états.

58. Optimiser l’évitement

Si le robot touche l’obstacle :

```
augmenter la distance de détection
```

```
réduire la vitesse d’approche
```

Si le robot contourne trop largement :

```
réduire la distance de contournement
```

Si le robot ne retrouve jamais la ligne :

```
modifier l’angle
```

```
modifier le mouvement de recherche
```

59. Optimiser la rotation

Si le robot dépasse l'angle :

```
réduire KP_ANGLE

réduire la correction maximum

augmenter légèrement la tolérance
```

Si le robot met trop de temps :

```
augmenter la correction maximum
```

tout en surveillant la précision.

60. Ajouter une vitesse adaptative

Une amélioration possible consiste à faire varier la vitesse pendant le suivi.

```
if abs(erreur) < 5:
    vitesse = 250
elif abs(erreur) < 20:
    vitesse = 180
else:
    vitesse = 100
```

Cela permet de :

```
accélérer en ligne droite

ralentir dans les courbes
```

61. Ajouter un PD ou PID

Si le suivi proportionnel n'est pas suffisamment stable, les étudiants avancés peuvent utiliser un PD ou un PID.

Par exemple :

```
p = erreur * kp

d = (
    erreur
    - erreur_precedente
) * kd

correction = p + d
```

Un PD peut être plus simple à régler qu'un PID complet.

62. Ne pas ajouter de complexité inutilement

Si :

```
P fonctionne parfaitement
```

il n'est pas nécessaire d'ajouter :

```
I et D
```

simplement pour rendre le programme plus complexe.

Le meilleur programme est celui qui répond correctement au besoin avec une complexité raisonnable.

63. Préparer la compétition

Chaque étudiant dispose d'un robot fonctionnel.

Avant la compétition :

```
vérifier la batterie
vérifier les câbles
vérifier les roues
vérifier les capteurs
recalibrer si nécessaire
```

Puis effectuer un dernier test de validation.

64. Règles possibles de compétition

Chaque équipe dispose de :

```
3 tentatives
```

Le meilleur résultat valide est conservé.

Une tentative valide signifie :

```
départ sans aide
mission terminée
arrivée détectée
aucune intervention humaine
```

65. Barème possible

Mission

```
40 points
```

Fiabilité

```
20 points
```

Temps

```
15 points
```

Qualité du code

```
10 points
```

Stratégie

```
10 points
```

Présentation

```
5 points
```

Total :

```
100 points
```

66. Exemple de détail du score

Critère	Points
Ligne trouvée	5
Suivi réussi	10
Obstacle détecté	5
Obstacle évité	10
Ligne retrouvée	5
Arrivée détectée	5
Fiabilité	20
Temps	15
Code	10
Stratégie	10
Présentation	5
Total	100

67. Pénalités possibles

Exemple :

```
collision

moins 2 points
intervention humaine

moins 10 points
robot remis sur la piste

tentative non valide
```

Les règles doivent être annoncées avant le début.

68. La vitesse ne doit pas être le seul critère

Un robot très rapide mais incapable de terminer la mission ne doit pas automatiquement gagner.

La compétition doit valoriser :

```
autonomie

fiabilité

qualité technique

vitesse
```

69. Présentation du projet

Après la compétition, chaque étudiant explique son robot.

Durée possible :

```
3 à 5 minutes
```

Tu dois présenter :

```
les capteurs

les états

la stratégie

la régulation

les principales difficultés

les améliorations réalisées
```

70. Questions possibles du professeur

Question 1

Pourquoi as-tu choisi cette distance de détection ?

Question 2

Pourquoi as-tu choisi ce `kp` ?

Question 3

Pourquoi ton robot utilise t-il ces états ?

Question 4

Quel est ton événement prioritaire ?

Question 5

Comment ton robot détecte t-il qu'il a perdu la ligne ?

Question 6

Que se passe t-il si l'ultrason ne détecte pas l'obstacle ?

Question 7

Quelle partie du programme a été la plus difficile à déboguer ?

Question 8

Que changerais tu avec une heure supplémentaire ?

71. Évaluation du code

Le professeur peut observer :

```
noms de variables clairs

fonctions courtes

paramètres regroupés

machine à états lisible

indentation correcte

absence de répétitions inutiles

gestion des erreurs
```

72. Exemple de mauvais code

```
if d < 200:
    x = 1
```

Les noms :

```
d
```

```
x
```

ne donnent presque aucune information.

73. Version plus claire

```
if obstacle_proche():
    etat = "RECULER"
```

Le programme devient presque lisible comme une phrase.

74. Évaluation de la stratégie

Deux robots peuvent utiliser des solutions différentes.

Robot A :

```
contourne l'obstacle avec plusieurs rotations précises
```

Robot B :

```
fait une grande courbe et recherche directement la ligne
```

Les deux solutions peuvent être valides si elles répondent au cahier des charges.

75. Il n'existe pas toujours une seule bonne solution

En programmation et en robotique, plusieurs stratégies peuvent résoudre le même problème.

L'important est de pouvoir expliquer :

```
pourquoi la solution a été choisie
```

```
quels sont ses avantages
```

```
quelles sont ses limites
```

76. Auto évaluation

Chaque étudiant complète :

```
Je sais créer une variable.

Oui / Presque / Non
Je sais utiliser if et elif.

Oui / Presque / Non
Je sais utiliser for et while.

Oui / Presque / Non
Je sais créer une fonction.

Oui / Presque / Non
Je sais lire un capteur.

Oui / Presque / Non
Je sais créer une machine à états.

Oui / Presque / Non
Je sais déboguer un programme.

Oui / Presque / Non
```

77. Questions finales de théorie de la première partie

Question 1

Quelle différence existe entre un capteur et un actionneur ?

Question 2

Pourquoi utilise t on des fonctions ?

Question 3

Quelle différence existe entre `if` et `while` ?

Question 4

Pourquoi calibrer un capteur ?

Question 5

Qu'est ce qu'une erreur de régulation ?

Question 6

Que fait la partie P d'un PID ?

Question 7

Que fait la partie I ?

Question 8

Que fait la partie D ?

Question 9

Qu'est ce qu'un état ?

Question 10

Qu'est ce qu'une transition ?

Question 11

Pourquoi définir des priorités ?

Question 12

Quelle est la première chose à faire lorsqu'un robot ne fonctionne pas ?

Correction

Réponse 1

Un capteur fournit une information au programme.
Un actionneur agit physiquement sur l'environnement.

Réponse 2

Pour organiser, réutiliser et simplifier le programme.

Réponse 3

`if` vérifie une condition.

`while` répète tant que la condition reste vraie.

Réponse 4

Parce que les valeurs dépendent du robot et de l'environnement.

Réponse 5

La différence entre une valeur souhaitée et une valeur mesurée.

Réponse 6

Elle corrige selon l'erreur actuelle.

Réponse 7

Elle tient compte de l'accumulation des erreurs.

Réponse 8

Elle tient compte de la variation de l'erreur.

Réponse 9

Une situation ou un comportement actuel du robot.

Réponse 10

Le passage d'un état à un autre lorsqu'une condition est remplie.

Réponse 11

Pour déterminer quel événement doit être traité en premier.

Réponse 12

Observer et décrire précisément le problème avant de modifier quoi que ce soit.

78. Bilan des 25 cours de la première partie

Bloc 1 : Python et MicroPython

Nous avons appris :

```
variables
types
calculs
conditions
while
for
```

Bloc 2 : déplacement

Nous avons appris :

```
moteurs
deux moteurs
DriveBase
fonctions
paramètres
projets de déplacement
```

Bloc 3 : capteurs

Nous avons appris :

```
tactile
ultrason
couleur
réflexion lumineuse
gyroscope
```

Bloc 4 : robotique avancée

Nous avons appris :

```
suivi de ligne
proportionnel
listes et historique
PID
```

machines à états

débogage

multi capteurs

priorités

Bloc 5 : projet

Nous avons appris :

concevoir

implémenter

tester

déboguer

optimiser

présenter

79. Ce que tu sais maintenant faire

À la fin de cette première partie, tu es capable de passer de :

J’ai un robot et un problème.

à :

J’analyse le problème.

Je choisis mes capteurs.

Je découpe le comportement.

Je crée mes fonctions.

Je définis mes états.

Je programme.

Je teste.

Je mesure.

Je corrige.

J’optimise.

C’est cette démarche qui est la compétence principale de la formation.

80. Projet supplémentaire possible

Créer un robot capable de sortir d’un labyrinthe.

Capteurs possibles :

```
ultrason
gyroscope
tactile
```

Notions :

```
machine à états
historique
navigation
```

81. Autre projet

Créer un robot de livraison.

Mission :

```
suivre une ligne
détecter différentes couleurs
s’arrêter à la bonne station
livrer un objet
reprendre le trajet
```

82. Autre projet

Créer une voiture autonome simplifiée.

Comportement :

```
suivre la route
respecter une distance
ralentir
éviter les obstacles
s’arrêter au rouge
```

Les notions utilisées seraient directement issues de ce cours.

83. Autre projet

Créer un robot qui suit une personne.

Utiliser :

```
ultrason
```

pour maintenir une distance.

Puis améliorer avec une régulation proportionnelle.

Conclusion générale de la première partie

Au début de la formation, le robot exécutait simplement :

```
BIP
```

Puis :

```
AVANCER
```

Ensuite :

```
SI événement

ALORS réaction
```

Puis :

```
MESURER
↓
CALCULER
↓
CORRIGER
```

Et finalement :

```
PERCEVOIR
↓
DÉCIDER
↓
AGIR
↓
ANALYSER LE RÉSULTAT
↓
S'ADAPTER
```

Le robot n'exécute plus seulement une suite d'instructions.

Il devient un système capable d'adapter son comportement à son environnement.

C'est le passage fondamental de la programmation classique vers la robotique autonome.

Deuxième partie

Cours 26 : Comprendre la connectivité de la brique EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Expliquer ce qu'est une connexion entre deux appareils
2. Identifier les différents moyens de communication de la brique EV3
3. Différencier USB, Bluetooth et WiFi
4. Comprendre la différence entre une connexion filaire et une connexion sans fil
5. Comprendre que l'EV3 fonctionne avec un système Linux appelé ev3dev
6. Identifier dans quels cas utiliser chaque type de connexion

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython
3. Un ordinateur
4. Un câble USB
5. Facultativement une deuxième brique EV3
6. Facultativement un adaptateur WiFi USB compatible

1. Introduction

Jusqu'à présent, nous avons surtout utilisé la brique EV3 comme un système autonome.

Le fonctionnement était généralement le suivant :

```
ORDINATEUR
    |
    | programme
    v
   EV3
    |
    v
MOTEURS ET CAPTEURS
```

Nous écrivons notre programme sur l'ordinateur.

Nous l'envoyons ensuite vers la brique EV3.

Enfin, l'EV3 exécute le programme.

Mais la brique EV3 peut également **communiquer avec d'autres appareils**.

Par exemple :

```
ORDINATEUR
    |
   USB
    |
    v
   EV3
```

Elle peut également communiquer sans fil :

```
MANETTE

   )))
   )))
Bluetooth
   )))
   )))

   EV3
```

Ou encore avec une autre brique :

```
EV3 1

  )))
  )))
Bluetooth
  )))
  )))

EV3 2
```

Nous allons donc commencer à transformer notre robot autonome en **robot connecté**.

2. Qu'est-ce qu'une connexion ?

Une connexion permet à deux appareils d'échanger des informations.

Imaginons deux personnes.

```
Alice  <==============>  Bob
```

Pour communiquer, elles doivent avoir un moyen de transmettre les informations.

Deux appareils fonctionnent de manière similaire.

```
APPAREIL A  <==============>  APPAREIL B
```

Les informations peuvent être transmises par :

```
un câble
```

ou

```
des ondes radio
```

3. Connexion filaire et connexion sans fil

Il existe deux grandes catégories.

Connexion filaire

Les appareils sont reliés physiquement.

Exemple :

```
PC ======== câble ======== EV3
```

L'USB est une connexion filaire.

Connexion sans fil

Aucun câble n'est nécessaire entre les deux appareils.

Exemple :

```
MANETTE  )))  )))  )))  EV3
```

Le Bluetooth est une connexion sans fil.

4. La connexion USB

L'EV3 possède un port USB.

Il permet notamment de relier la brique à un ordinateur.

```
ORDINATEUR

     |
     |
    USB
     |
     |
     v

    EV3
```

À quoi sert l'USB ?

Nous pouvons l'utiliser pour :

1. envoyer des programmes vers l'EV3
2. communiquer avec la brique
3. accéder à la brique depuis l'ordinateur
4. établir une connexion réseau USB avec ev3dev

Avantages

L'USB est généralement :

```
stable
rapide à mettre en place
simple à dépanner
```

Inconvénient

Le robot reste relié à l'ordinateur par un câble.

Ce n'est donc pas idéal pour un robot qui doit se déplacer librement.

5. Le Bluetooth

La brique EV3 possède directement un module Bluetooth.

Aucun adaptateur supplémentaire n'est nécessaire.

Bluetooth utilise des ondes radio pour transmettre les informations.

```
EV3

 ))))))))))))))))

       Bluetooth

 ))))))))))))))))

ORDINATEUR
```

Le Bluetooth peut être utilisé avec différents appareils.

Par exemple :

```
EV3  )))  ORDINATEUR
EV3  )))  EV3
EV3  )))  MANETTE
```

C'est cette dernière possibilité que nous utiliserons plus tard pour piloter notre robot avec une manette sans fil.

6. Le WiFi

Contrairement au Bluetooth, la brique EV3 ne possède pas directement de WiFi intégré.

Pour utiliser un réseau WiFi, il faut généralement ajouter un adaptateur WiFi USB compatible avec ev3dev.

```
EV3
 |
 |
ADAPTATEUR
WiFi USB
 |
 )))
 )))
 )))
 |
ROUTEUR WiFi
```

Le WiFi devient particulièrement intéressant lorsque l'on souhaite connecter l'EV3 à un véritable réseau informatique.

7. Comparaison USB, Bluetooth et WiFi

Technologie	Câble nécessaire	Intégré à l'EV3	Utilisation typique
USB	Oui	Oui	Programmation et connexion PC
Bluetooth	Non	Oui	EV3, ordinateur, manette
WiFi	Non	Non	Réseau informatique

8. Quel type choisir ?

Imagine que tu souhaites programmer ton robot depuis ton ordinateur.

Le choix le plus simple est souvent :

```
USB
```

Imagine maintenant que tu souhaites piloter ton robot avec une manette.

Le choix devient :

```
Bluetooth
```

Enfin, si tu veux intégrer ton EV3 dans un réseau informatique :

```
WiFi
```

peut être très intéressant.

9. Mais qu'est réellement notre EV3 ?

Voici une notion importante.

Notre brique EV3 n'exécute pas uniquement MicroPython.

Elle utilise également un véritable système d'exploitation.

Dans notre environnement :

```
PROGRAMME MICROPYTHON
        |
        v
     PYBRICKS
        |
        v
     ev3dev
        |
        v
      LINUX
        |
        v
   BRIQUE EV3
```

Linux

Linux est un système d'exploitation.

C'est le même type de système que l'on peut retrouver sur :

```
des serveurs
des ordinateurs
des Raspberry Pi
des systèmes embarqués
```

Notre EV3 est donc en réalité un **petit ordinateur Linux**.

10. Pourquoi est-ce important ?

Jusqu'à présent, nous avons surtout utilisé Pybricks.

Par exemple :

```
from pybricks.ev3devices import Motor
```

Pybricks simplifie énormément l'utilisation du matériel LEGO.

Mais certaines fonctionnalités plus avancées ne sont pas directement gérées par Pybricks.

C'est notamment le cas de certaines manettes Bluetooth.

Dans ce cas, nous pourrons utiliser directement les fonctions disponibles dans Linux.

Nous aurons donc deux niveaux.

```
NIVEAU SIMPLE

MicroPython
    |
Pybricks
    |
EV3
```

et :

```
NIVEAU AVANCE

Python
    |
Linux
    |
Périphérique Bluetooth
```

11. Exemple avec une manette

Lorsque nous appuierons plus tard sur un bouton de la manette :

```
BOUTON X
   |
   v
MANETTE
   |
Bluetooth
   |
   v
EV3
   |
   v
LINUX
   |
   v
PROGRAMME PYTHON
   |
   v
MOTEUR
```

Notre programme devra donc récupérer l'information provenant de la manette puis décider quoi faire.

Par exemple :

```
X appuyé
    |
    v
avancer
```

ou :

```
Carré appuyé
      |
      v
     STOP
```

12. Activité 1 : reconnaître les connexions

Indique s’il s’agit d’une connexion filaire ou sans fil.

A

```
PC ===== USB ===== EV3
```

Réponse :

```
Filaire
```

B

```
MANETTE ))) EV3
```

Réponse :

```
Sans fil
```

C

```
EV3 ))) ROUTEUR
```

Réponse :

```
Sans fil
```

13. Activité 2 : choisir la bonne technologie

Pour chaque situation, choisis entre :

```
USB
Bluetooth
WiFi
```

Situation 1

Je veux relier rapidement mon EV3 à mon ordinateur avec un câble.

Réponse :

```
USB
```

Situation 2

Je veux contrôler mon robot avec une manette.

Réponse :

```
Bluetooth
```

Situation 3

Je veux connecter plusieurs appareils sur mon réseau informatique.

Réponse :

```
WiFi
```

Situation 4

Je veux faire communiquer deux briques EV3 sans câble.

Réponse :

```
Bluetooth
```

14. Manipulation sur la brique

Allume la brique EV3.

Dans le menu ev3dev, recherche la section consacrée aux connexions sans fil.

Tu devrais trouver les paramètres Bluetooth.

L'objectif aujourd'hui est uniquement de les identifier.

Repère notamment :

```
Bluetooth
Powered
Visible
Start Scan
```

Powered

Cette option permet d'activer ou désactiver Bluetooth.

```
Powered = ON
```

signifie que le Bluetooth est actif.

Visible

Cette option permet aux autres appareils de voir la brique.

```
Visible = ON
```

Start Scan

Cette option permet à l'EV3 de rechercher les appareils Bluetooth présents autour de lui.

```
Start Scan
      |
      v
Recherche
      |
      v
Téléphones
ordinateurs
autres EV3
manettes
```

15. Mini défi

Sans connecter d'appareil, trouve les trois éléments suivants :

```
Powered
Visible
Start Scan
```

Explique à ton voisin ce que chacun signifie.

16. Défi 1

Complète le schéma suivant.

```
MANETTE

     ?

    EV3
```

Réponse :

```
MANETTE

   Bluetooth

     EV3
```

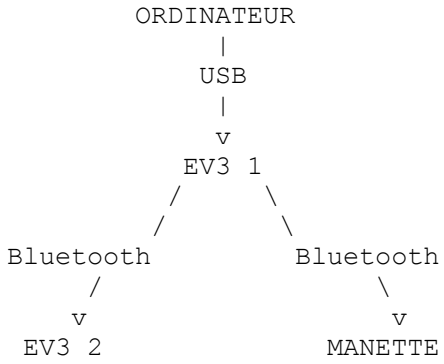
17. Défi 2

On possède :

```
1 ordinateur
2 briques EV3
1 manette
```

Propose un système de connexion possible.

Exemple :



Il n'existe pas forcément une seule bonne réponse.

L'important est de pouvoir expliquer son choix.

18. Défi 3 : quelle connexion utiliser ?

Un robot doit participer à une course.

Il doit être piloté à distance et aucun câble ne peut être attaché au robot.

Quelle technologie choisirais tu ?

Réponse attendue :

```
Bluetooth
```

Pourquoi ?

Parce qu'elle permet une connexion sans fil et qu'elle est directement intégrée à l'EV3.

19. Questions de compréhension

Question 1

Qu'est-ce qu'une connexion ?

Question 2

Quelle technologie nécessite un câble ?

Question 3

Quelle technologie sans fil est directement intégrée à la brique EV3 ?

Question 4

La brique EV3 possède t elle directement du WiFi ?

Question 5

Que faut il ajouter pour utiliser généralement le WiFi avec l'EV3 ?

Question 6

Qu'est ce qu'ev3dev ?

Question 7

Pourquoi est il intéressant de savoir que l'EV3 fonctionne sous Linux ?

Correction 1

Un moyen permettant à deux appareils d'échanger des informations.

Correction 2

USB.

Correction 3

Bluetooth.

Correction 4

Non.

Correction 5

Un adaptateur WiFi USB compatible.

Correction 6

Un système Linux conçu notamment pour faire fonctionner la brique EV3.

Correction 7

Parce que cela permet d'utiliser des fonctionnalités plus avancées, notamment certains périphériques comme les manettes.

20. Petit exercice écrit

Complète les phrases.

```
USB est une connexion __________.

Bluetooth est une connexion __________.

Pour connecter directement une manette à l'EV3, nous utiliserons principalement
__________.

Le système d'exploitation utilisé par notre EV3 est basé sur __________.
```

Correction

```
USB est une connexion filaire.

Bluetooth est une connexion sans fil.

Pour connecter directement une manette à l'EV3, nous utiliserons principalement
Bluetooth.

Le système d'exploitation utilisé par notre EV3 est basé sur Linux.
```

21. Activité facultative : observer les connexions réseau

Si l'EV3 est connecté à l'ordinateur, l'enseignant peut montrer qu'il possède également une identité sur le réseau.

On peut introduire simplement le concept :

```
ORDINATEUR

192.168.x.x

      |
      |
    réseau
      |
      |

EV3

192.168.x.x
```

Il n'est pas nécessaire de comprendre les adresses IP à ce stade.

Nous y reviendrons dans la leçon 35.

22. Erreurs fréquentes

Erreur 1

Penser que Bluetooth signifie automatiquement Internet.

Correction :

Bluetooth permet simplement à des appareils proches de communiquer.

Erreur 2

Confondre Bluetooth et WiFi.

Correction :

Les deux utilisent des communications sans fil, mais ce sont des technologies différentes.

Erreur 3

Penser que MicroPython remplace complètement Linux.

Correction :

Dans notre configuration, MicroPython fonctionne au dessus d'ev3dev Linux.

Erreur 4

Penser que l'EV3 possède directement du WiFi.

Correction :

Il possède directement Bluetooth, mais un adaptateur est nécessaire pour le WiFi.

Cours 27 : Associer deux briques EV3 en Bluetooth

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Activer le Bluetooth sur une brique EV3
2. Rendre une brique visible
3. Rechercher un autre appareil Bluetooth
4. Associer deux briques EV3
5. Comprendre la différence entre association et connexion
6. Identifier le rôle du serveur et du client pour la suite

Matériel nécessaire

1. Deux briques LEGO Education Mindstorms EV3
2. Deux cartes microSD avec EV3 MicroPython
3. Deux batteries chargées
4. Facultativement deux ordinateurs pour faciliter les manipulations

1. Introduction

Lors de la leçon précédente, nous avons découvert plusieurs moyens de connecter la brique EV3.

Nous allons maintenant réellement utiliser le Bluetooth.

L'objectif est d'obtenir ceci :

```
EV3 A

   )))
   )))
Bluetooth
   )))
   )))

EV3 B
```

Mais avant que deux appareils puissent communiquer, ils doivent d'abord se reconnaître.

Cette opération s'appelle :

```
PAIRING
```

ou en français :

```
ASSOCIATION
```

2. Qu’est-ce que l’association Bluetooth ?

Prenons l’exemple d’un téléphone et d’écouteurs Bluetooth.

La première fois que nous utilisons les écouteurs, nous devons les associer au téléphone.

```
TELEPHONE

     cherche

       )))

       )))

    ECOUTEURS
```

Lorsque l’association est terminée, les deux appareils se connaissent.

La prochaine fois, ils pourront plus facilement se reconnecter.

Avec deux briques EV3, le principe est identique.

```
EV3 A

     cherche

       )))

       )))

     EV3 B
```

3. Association et connexion

Il est très important de ne pas confondre ces deux notions.

Association

L’association permet aux appareils de se reconnaître.

On peut voir cela comme :

```
"Je connais cet appareil."
```

Connexion

La connexion correspond au moment où les deux appareils communiquent réellement.

```
"Je communique maintenant avec cet appareil."
```

On peut donc avoir :

```
ASSOCIE
```

mais :

```
PAS CONNECTE
```

à un instant donné.

4. Exemple simple

Ton téléphone connaît probablement plusieurs appareils Bluetooth.

Par exemple :

```
telephone
 |
 | connait
 |
 + casque
 |
 + voiture
 |
 + enceinte
```

Mais il n’est pas forcément connecté aux trois en même temps.

C’est exactement la même logique avec l’EV3.

5. Préparer les deux briques

Nous allons appeler les briques :

```
EV3 A
```

et :

```
EV3 B
```

Allume les deux briques.

Attends que le système ev3dev soit complètement démarré.

6. Activer le Bluetooth

Sur les deux briques, ouvre le menu Bluetooth.

Tu dois trouver une option semblable à :

```
Wireless and Networks
```

puis :

```
Bluetooth
```

Active :

```
Powered
```

Le Bluetooth est maintenant actif.

7. Rendre les briques visibles

Active ensuite :

```
Visible
```

Cela permet aux autres appareils Bluetooth présents à proximité de voir la brique.

On peut représenter cela comme ceci :

```
EV3 B

"Je suis visible"

     )))
     )))
     )))
```

Si `Visible` est désactivé, l'autre brique peut ne pas la trouver lors de la recherche.

8. Rechercher un appareil

Sur EV3 A, sélectionne :

```
Start Scan
```

La brique commence à rechercher les périphériques Bluetooth proches.

```
EV3 A

    cherche

  )))  )))  )))

      EV3 B
```

Après quelques secondes, plusieurs appareils peuvent apparaître.

Par exemple :

```
ev3dev
Wireless Controller
Telephone
Laptop
```

Dans notre cas, nous devons identifier EV3 B.

9. Un problème possible : les noms identiques

Si plusieurs briques utilisent le nom :

```
ev3dev
```

il devient très difficile de savoir laquelle choisir.

Imagine une classe avec huit briques :

```
ev3dev
ev3dev
ev3dev
ev3dev
ev3dev
ev3dev
ev3dev
ev3dev
```

Ce n’est pas très pratique.

Il est donc utile de donner un nom différent à chaque brique.

Par exemple :

```
EV3_ROUGE
```

et :

```
EV3_BLEU
```

Ou simplement :

```
EV3_A
```

et :

```
EV3_B
```

10. Pourquoi nommer les briques ?

Le nom sert d'identifiant lisible pour l'utilisateur.

Par exemple :

```
EV3_PROF
EV3_01
EV3_02
EV3_03
```

Cela facilite énormément les manipulations en classe.

Pour les prochaines leçons, il est recommandé de conserver des noms simples et uniques.

11. Associer les deux briques

Une fois EV3 B trouvée dans la liste :

1. Sélectionne EV3 B
2. Choisis l'option d'association
3. Accepte la demande sur l'autre brique si nécessaire
4. Vérifie que l'appareil apparaît comme connu ou associé

À la fin, nous voulons obtenir :

```
EV3 A
   |
   |
associe avec
   |
   |
EV3 B
```

12. Attention à la connexion manuelle

Pour la communication MicroPython entre deux briques, il n'est généralement pas nécessaire de connecter manuellement les deux briques depuis le menu après l'association.

Le programme Bluetooth se chargera ensuite d'établir la connexion.

Il faut donc surtout vérifier que les deux appareils sont correctement associés.

13. Vérification

Sur EV3 A, regarde la liste des appareils Bluetooth connus.

EV3 B doit apparaître.

Fais ensuite la même vérification sur EV3 B.

L'objectif est que les deux briques se reconnaissent correctement.

14. Activité 1 : observer les états

Pour chaque situation, indique si les appareils sont associés, connectés ou les deux.

Situation A

Les deux briques se sont déjà reconnues mais aucun programme ne fonctionne.

Réponse attendue :

```
ASSOCIEES
```

mais pas nécessairement connectées.

Situation B

Un programme Bluetooth est en train d'échanger des messages.

Réponse :

```
ASSOCIEES
ET
CONNECTEES
```

15. Activité 2 : trouver la bonne brique

L'enseignant allume plusieurs EV3.

Chaque étudiant doit identifier sa propre brique dans la liste Bluetooth.

Exemple :

```
EV3_01
EV3_02
EV3_03
EV3_04
```

Chaque étudiant doit sélectionner le bon appareil.

16. Activité 3 : dessiner la procédure

Replace les étapes dans le bon ordre :

```
Associer l'appareil

Activer Bluetooth

Scanner les appareils

Rendre la brique visible
```

Correction

```
1. Activer Bluetooth

2. Rendre la brique visible

3. Scanner les appareils

4. Associer l'appareil
```

17. Introduction au serveur et au client

Dans la prochaine leçon, nous allons faire communiquer réellement les deux briques.

Pour cela, nous allons utiliser deux rôles.

```
SERVEUR
```

et :

```
CLIENT
```

Le serveur

Le serveur attend une connexion.

```
EV3 A

SERVEUR

"J'attends"
```

Le client

Le client démarre la connexion.

```
EV3 B

CLIENT

"Je me connecte"
```

18. Analogie simple

Imagine un restaurant.

Le serveur est présent et attend les clients.

Le client arrive et demande à être servi.

Pour notre communication Bluetooth :

```
EV3 SERVEUR

   attend

     ^

     |

 connexion

     |

EV3 CLIENT
```

19. Est-ce toujours le même EV3 qui doit être serveur ?

Non.

Nous pouvons décider que :

```
EV3 A = serveur
```

et :

```
EV3 B = client
```

Mais nous pourrions également inverser les rôles.

Le choix dépend simplement de l'organisation du programme.

20. Préparation de la prochaine leçon

Pour faciliter la suite, définissons maintenant :

```
EV3 A = SERVEUR
```

et :

```
EV3 B = CLIENT
```

Note également le nom Bluetooth exact du serveur.

Par exemple :

`EV3_A`

Ce nom sera utilisé dans notre code MicroPython.

21. Mini défi 1

Associe correctement EV3 A et EV3 B.

Puis explique à ton voisin la différence entre :

`VISIBLE`

et :

`ASSOCIE`

Correction

`Visible` signifie que la brique peut être découverte par d'autres appareils.

`Associé` signifie que les deux appareils se connaissent déjà.

22. Mini défi 2

Un camarade te dit :

J'ai associé les deux EV3, donc ils échangent forcément des messages.

Cette phrase est-elle correcte ?

Réponse

Non.

L'association prépare les appareils à communiquer.

Pour réellement échanger des données, nous devons établir une connexion et exécuter un programme.

23. Défi pratique

Deux étudiants collaborent pour ce défi avec leur brique EV3.

L'objectif est de :

1. activer Bluetooth
2. rendre les deux briques visibles
3. identifier leur nom
4. rechercher l'autre brique
5. effectuer l'association
6. vérifier que l'association est enregistrée

Lorsque tout fonctionne, le groupe doit pouvoir expliquer chaque étape sans regarder le cours.

24. Défi supplémentaire

Éteignez les deux EV3.

Rallumez les ensuite.

Observez si l'association Bluetooth est toujours enregistrée.

Question :

Doit-on refaire toute l'association ?

Dans la majorité des cas, non.

L'association est enregistrée.

25. Situation problème

Imagine que vous effectuez :

```
Start Scan
```

mais que EV3 B n'apparaît pas.

Quelles sont les causes possibles ?

Réponses possibles :

```
Bluetooth désactivé
EV3 B pas visible
EV3 B éteinte
mauvais appareil sélectionné
problème temporaire Bluetooth
```

26. Méthode de dépannage

Lorsqu'une connexion Bluetooth ne fonctionne pas, il est utile de suivre un ordre précis.

```
1. La brique est-elle allumée ?

2. Bluetooth est-il activé ?

3. La brique est-elle visible ?

4. Le scan trouve t-il l'appareil ?

5. L'association est-elle réalisée ?

6. Le nom de l'appareil est-il correct ?
```

Cette méthode est importante car elle introduit une vraie logique de dépannage informatique.

27. Exercice de dépannage

Situation :

EV3 A ne trouve pas EV3 B.

Informations :

```
EV3 A
Powered = ON

EV3 B
Powered = ON
Visible = OFF
```

Quel est probablement le problème ?

Correction

EV3 B n'est pas visible.

Il faut activer :

```
Visible
```

28. Deuxième exercice de dépannage

Les deux briques sont associées correctement.

Mais le programme de la prochaine leçon indique qu'il ne trouve pas :

```
EV3_SERVEUR
```

Le nom réel de la brique est :

`EV3_A`

Quelle est l'erreur ?

Correction

Le programme utilise un mauvais nom Bluetooth.

29. Questions de compréhension

Question 1

Que signifie associer deux appareils Bluetooth ?

Question 2

Que signifie `Powered` ?

Question 3

Que signifie `Visible` ?

Question 4

À quoi sert `Start Scan` ?

Question 5

Une brique associée est elle forcément connectée ?

Question 6

Quel appareil démarre généralement la connexion dans une architecture client serveur ?

Question 7

Quel appareil attend généralement la connexion ?

Correction 1

Cela signifie qu'ils se reconnaissent et enregistrent les informations nécessaires pour pouvoir communiquer.

Correction 2

Le Bluetooth est activé.

Correction 3

Les autres appareils peuvent découvrir la brique.

Correction 4

À rechercher les appareils Bluetooth proches.

Correction 5

Non.

Correction 6

Le client.

Correction 7

Le serveur.

30. Petit exercice écrit

Complète les phrases suivantes.

```
Avant de communiquer, deux appareils Bluetooth doivent être __________.

L'option __________ permet d'activer Bluetooth.

L'option __________ permet aux autres appareils de voir la brique.

L'option __________ permet de rechercher les périphériques proches.

Le __________ démarre la connexion.

Le __________ attend la connexion.
```

Correction

```
Avant de communiquer, deux appareils Bluetooth doivent être associés.

L'option Powered permet d'activer Bluetooth.

L'option Visible permet aux autres appareils de voir la brique.

L'option Start Scan permet de rechercher les périphériques proches.

Le client démarre la connexion.

Le serveur attend la connexion.
```

31. Erreurs fréquentes

Erreur 1

Bluetooth est désactivé sur une des deux briques.

Solution :

Vérifier `Powered`.

Erreur 2

La brique distante n’apparaît pas.

Solution :

Vérifier `Visible`.

Erreur 3

Plusieurs EV3 ont le même nom.

Solution :

Attribuer un nom unique à chaque brique.

Erreur 4

Tu pourrais penser qu’associer suffit pour envoyer des données.

Solution :

Rappeler que l’association prépare la communication mais qu’un programme doit ensuite établir la connexion.

Erreur 5

Le mauvais EV3 est choisi pendant le scan.

Solution :

Identifier précisément le nom des briques avant l’association.

Cours 28 : Envoyer des messages entre deux briques EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Comprendre le principe d'une communication client serveur
2. Utiliser `BluetoothMailboxServer`
3. Utiliser `BluetoothMailboxClient`
4. Créer une `TextMailbox`
5. Envoyer un message Bluetooth
6. Recevoir un message Bluetooth
7. Faire répondre une deuxième brique EV3
8. Diagnostiquer les erreurs de connexion les plus fréquentes

Matériel nécessaire

1. Deux briques LEGO Education Mindstorms EV3
2. Deux cartes microSD avec EV3 MicroPython
3. Deux ordinateurs, si possible
4. Les deux briques déjà associées en Bluetooth
5. Le nom Bluetooth exact de la brique serveur

1. Introduction

Lors de la leçon précédente, nous avons associé deux briques EV3.

Nous avions :

```
EV3 A
SERVEUR

   )))

Bluetooth

   )))

EV3 B
CLIENT
```

Mais aucune donnée n'était encore envoyée.

Aujourd'hui, nous allons envoyer notre premier message.

L’objectif sera :

```
EV3 CLIENT

"Bonjour"

      |
      |
  Bluetooth
      |
      v

EV3 SERVEUR

"Bonjour"
```

2. Rappel : serveur et client

Nous allons conserver les rôles suivants :

```
EV3 A = SERVEUR
EV3 B = CLIENT
```

Le serveur attend.

Le client lance la connexion.

```
SERVEUR

"J'attends une connexion"

          ^

          |
      Bluetooth
          |

          |

CLIENT

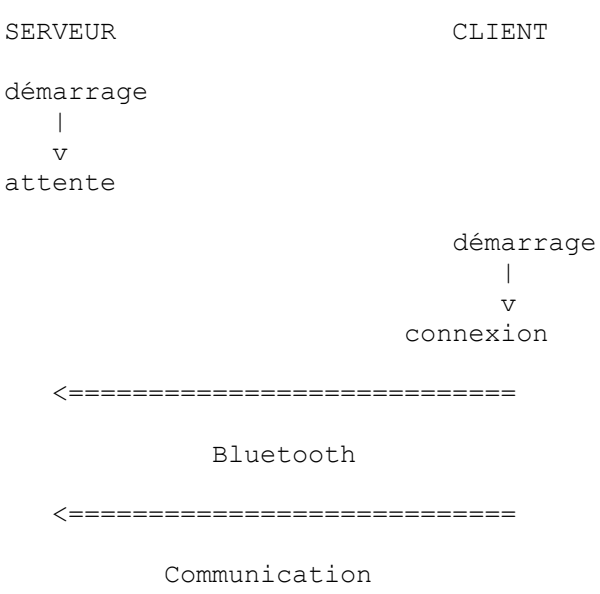
"Je me connecte"
```

3. Principe de fonctionnement

Notre communication suivra quatre étapes.

```
1. Le serveur démarre
2. Le serveur attend
3. Le client se connecte
4. Les deux briques échangent des messages
```

Schéma :



4. Qu’est-ce qu’une Mailbox ?

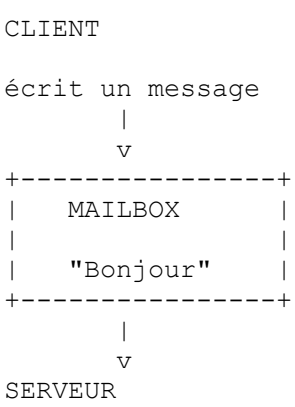
Pybricks utilise le concept de :

```
Mailbox
```

On peut traduire cela par :

```
boîte aux lettres
```

Imaginons une vraie boîte aux lettres.



Une mailbox contient une information que l’autre EV3 peut recevoir.

5. Nom de la Mailbox

Chaque mailbox possède un nom.

Par exemple :

```
TextMailbox("message", ...)
```

Le nom est ici :

```
message
```

Il est extrêmement important que les deux briques utilisent exactement le même nom.

Serveur :

```
TextMailbox("message", server)
```

Client :

```
TextMailbox("message", client)
```

Si l'un utilise :

```
message
```

et l'autre :

```
messages
```

la communication ne fonctionnera pas comme prévu.

6. Les différents types de Mailbox

Pybricks propose plusieurs types de mailbox.

Les plus utiles sont :

```
TextMailbox
```

pour du texte.

```
NumericMailbox
```

pour des nombres.

```
LogicMailbox
```

pour des valeurs logiques.

Par exemple :

```
TextMailbox
```

```
"Bonjour"
```

```
NumericMailbox

75
LogicMailbox

True
```

Aujourd'hui, nous allons commencer par `TextMailbox`.

7. Programme du serveur

Sur EV3 A, crée un nouveau programme.

```
#!/usr/bin/env pybricks-micropython

from pybricks.messaging import BluetoothMailboxServer
from pybricks.messaging import TextMailbox

server = BluetoothMailboxServer()

mailbox = TextMailbox("message", server)

print("Serveur demarre")
print("Attente du client...")

server.wait_for_connection()

print("Client connecte")

mailbox.wait()

message = mailbox.read()

print("Message recu :")
print(message)
```

8. Comprendre le programme serveur

Regardons chaque partie.

Importation

```
from pybricks.messaging import BluetoothMailboxServer
```

Cette ligne permet de créer un serveur Bluetooth.

Puis :

```
from pybricks.messaging import TextMailbox
```

nous permet de créer une mailbox contenant du texte.

9. Création du serveur

```
server = BluetoothMailboxServer()
```

Nous créons un objet appelé :

```
server
```

Cet objet représente notre serveur Bluetooth.

10. Création de la Mailbox

```
mailbox = TextMailbox("message", server)
```

Nous créons une boîte aux lettres.

Son nom est :

```
message
```

Elle appartient au serveur.

11. Attendre le client

```
server.wait_for_connection()
```

Cette instruction bloque le programme.

Cela signifie que le programme reste ici tant qu'un client ne se connecte pas.

```
PROGRAMME

      |
      v

wait_for_connection()

      |
      |
      | attend
      |
      |
      v

CLIENT CONNECTE
```

12. Attendre un message

Après la connexion :

```
mailbox.wait()
```

Le serveur attend maintenant qu'un nouveau message arrive.

```
CLIENT

     |
     | envoie
     v

MAILBOX

     |
     v

mailbox.wait()
```

13. Lire le message

Lorsque le message est arrivé :

```
message = mailbox.read()
```

Le contenu de la mailbox est placé dans la variable :

```
message
```

Puis :

```
print(message)
```

affiche son contenu.

14. Programme du client

Sur EV3 B, crée un deuxième programme.

```
#!/usr/bin/env pybricks-micropython

from pybricks.messaging import BluetoothMailboxClient
from pybricks.messaging import TextMailbox

client = BluetoothMailboxClient()

mailbox = TextMailbox("message", client)
print("Connexion au serveur...")

client.connect("EV3_A")

print("Connexion reussie")

mailbox.send("Bonjour EV3")
print("Message envoye")
```

Il faut remplacer :

```
EV3_A
```

par le nom Bluetooth exact de ta brique serveur.

15. Comprendre le client

La première différence est :

```
BluetoothMailboxClient
```

Cette fois, notre programme n’attend pas.

Il va rechercher le serveur.

16. Connexion au serveur

Cette ligne est essentielle :

```
client.connect("EV3_A")
```

Elle indique :

```
Connecte toi au périphérique appelé EV3_A.
```

Le nom doit être exactement correct.

Par exemple, si le serveur s’appelle réellement :

```
EV3_01
```

il faut écrire :

```
client.connect("EV3_01")
```

17. Envoyer le message

Cette instruction :

```
mailbox.send("Bonjour EV3")
```

envoie le texte :

```
Bonjour EV3
```

vers la mailbox.

18. Ordre de démarrage

Il faut respecter l’ordre suivant.

D’abord :

```
SERVEUR
```

Puis :

```
CLIENT
```

Pourquoi ?

Parce que le serveur doit déjà être en train d’attendre lorsque le client tente de se connecter.

19. Première manipulation

Sur EV3 A :

lance le programme serveur.

Tu dois obtenir :

```
Serveur demarre
Attente du client...
```

Le programme reste bloqué.

C’est normal.

Il attend.

Ensuite, lance le programme client sur EV3 B.

Le client devrait afficher :

```
Connexion au serveur...
Connexion reussie
Message envoye
```

Le serveur devrait ensuite afficher :

```
Client connecte
Message recu :
Bonjour EV3
```

Bravo.

Tu viens d’envoyer ton premier message Bluetooth entre deux EV3.

20. Représentation complète

```
EV3 B
CLIENT

mailbox.send("Bonjour")

          |
          |
          v

      Bluetooth

          |
          |
          v

EV3 A
SERVEUR

mailbox.wait()

          |
          v

mailbox.read()

          |
          v

"Bonjour"
```

21. Exercice 1

Modifie le programme client afin d'envoyer :

```
Bonjour, je suis le robot B
```

Résultat attendu sur le serveur :

```
Message recu :
Bonjour, je suis le robot B
```

22. Exercice 2

Envoie ton prénom.

Par exemple :

```
mailbox.send("Quentin")
```

Le serveur doit afficher :

```
Message recu :
Quentin
```

23. Exercice 3

Créer une variable avant l'envoi.

```
message = "Bonjour EV3"
```

Puis :

```
mailbox.send(message)
```

Programme :

```
message = "Bonjour EV3"

mailbox.send(message)
```

Question :

Est-ce que le résultat est différent ?

Réponse :

Non.

Nous avons simplement stocké le texte dans une variable avant de l'envoyer.

24. Faire répondre le serveur

Pour l'instant, la communication se fait dans un seul sens.

```
CLIENT

   |
   | message
   v

SERVEUR
```

Mais le serveur peut également envoyer des informations.

Nous voulons :

```
CLIENT

"Bonjour"

    |
    v

SERVEUR

"Message recu"

    |
    v

CLIENT
```

25. Programme serveur avec réponse

Modifie le serveur :

```
#!/usr/bin/env pybricks-micropython

from pybricks.messaging import BluetoothMailboxServer
from pybricks.messaging import TextMailbox

server = BluetoothMailboxServer()

mailbox = TextMailbox("message", server)

print("Attente du client...")

server.wait_for_connection()

print("Client connecte")

mailbox.wait()

message = mailbox.read()

print("Message recu :")
print(message)

mailbox.send("Message recu")
```

26. Programme client avec réception

Modifie maintenant le client.

```
#!/usr/bin/env pybricks-micropython

from pybricks.messaging import BluetoothMailboxClient
from pybricks.messaging import TextMailbox

client = BluetoothMailboxClient()

mailbox = TextMailbox("message", client)

client.connect("EV3_A")

mailbox.send("Bonjour EV3")

print("Message envoye")

mailbox.wait()

reponse = mailbox.read()

print("Reponse du serveur :")
print(reponse)
```

27. Résultat attendu

Client :

```
Message envoye

Reponse du serveur :
Message recu
```

Serveur :

```
Client connecte

Message recu :
Bonjour EV3
```

La communication fonctionne maintenant dans les deux sens.

28. Important : attendre un nouveau message

La méthode :

```
mailbox.wait()
```

attend une modification de la mailbox.

Elle est donc très pratique lorsque l'on souhaite synchroniser deux programmes.

Cela évite de lire continuellement la mailbox sans savoir si un nouveau message est arrivé.

29. Mini défi 1

Créer la conversation suivante.

Client :

```
Bonjour
```

Serveur :

```
Salut
```

Client :

```
Comment vas tu ?
```

Serveur :

```
Tres bien
```

30. Une solution possible

Serveur :

```
#!/usr/bin/env pybricks-micropython

from pybricks.messaging import BluetoothMailboxServer
from pybricks.messaging import TextMailbox

server = BluetoothMailboxServer()
mailbox = TextMailbox("message", server)

server.wait_for_connection()

mailbox.wait()
print(mailbox.read())

mailbox.send("Salut")

mailbox.wait()
print(mailbox.read())

mailbox.send("Tres bien")
```

Client :

```
#!/usr/bin/env pybricks-micropython

from pybricks.messaging import BluetoothMailboxClient
from pybricks.messaging import TextMailbox

client = BluetoothMailboxClient()
mailbox = TextMailbox("message", client)

client.connect("EV3_A")

mailbox.send("Bonjour")

mailbox.wait()
print(mailbox.read())

mailbox.send("Comment vas tu ?")

mailbox.wait()
print(mailbox.read())
```

31. Exercice : prédire le résultat

Observe :

```
mailbox.send("ROUGE")
```

Que recevra l'autre EV3 ?

Réponse :

```
ROUGE
```

Et avec :

```
mailbox.send("123")
```

Réponse :

Le texte :

```
123
```

Même si cela ressemble à un nombre, une `TextMailbox` traite cette valeur comme du texte.

32. NumericMailbox

Pour envoyer de vrais nombres, nous pouvons utiliser :

```
NumericMailbox
```

Par exemple :

```
from pybricks.messaging import NumericMailbox
```

Puis :

```
mailbox = NumericMailbox("speed", client)
```

et :

```
mailbox.send(75)
```

L'autre EV3 peut récupérer :

```
speed = mailbox.read()
```

et utiliser directement cette valeur dans un calcul.

Cette notion sera très utile lors de la prochaine leçon pour contrôler la vitesse des moteurs.

33. Exemple conceptuel

```
EV3 TELECOMMANDE

envoie 80
      |
      v
Bluetooth
      |
      v
EV3 ROBOT

speed = 80
      |
      v
MOTEURS
```

Nous préparons donc déjà la leçon 29.

34. LogicMailbox

Une `LogicMailbox` permet d'envoyer :

```
True
```

ou :

```
False
```

Cela peut être utile pour transmettre un état.

Par exemple :

```
obstacle détecté = True
```

ou :

```
bouton appuyé = False
```

35. Mini défi 2

Créer une `NumericMailbox`.

Le client doit envoyer :

```
50
```

Le serveur doit afficher la valeur reçue.

Exemple serveur :

```
from pybricks.messaging import BluetoothMailboxServer
from pybricks.messaging import NumericMailbox

server = BluetoothMailboxServer()

mailbox = NumericMailbox("number", server)

server.wait_for_connection()

mailbox.wait()

number = mailbox.read()
print(number)
```

Client :

```
from pybricks.messaging import BluetoothMailboxClient
from pybricks.messaging import NumericMailbox

client = BluetoothMailboxClient()

mailbox = NumericMailbox("number", client)

client.connect("EV3_A")

mailbox.send(50)
```

36. Défi 3

Le client envoie deux nombres :

```
25
```

puis :

```
75
```

Le serveur doit les afficher l'un après l'autre.

Résultat attendu :

```
25
75
```

37. Défi final : mini système de commande

Le client doit envoyer des commandes sous forme de texte.

```
AVANCE
RECULE
STOP
```

Le serveur affiche la commande reçue.

Exemple :

```
commande = mailbox.read()

if commande == "AVANCE":
    print("Le robot doit avancer")

elif commande == "RECULE":
    print("Le robot doit reculer")

elif commande == "STOP":
    print("Le robot doit s'arreter")
```

Pour l'instant, aucun moteur ne bouge.

L'objectif est uniquement de comprendre la transmission des commandes.

38. Architecture du défi

```
EV3 CLIENT

bouton haut
     |
     v
"AVANCE"
     |
     v
Bluetooth
     |
     v

EV3 SERVEUR

commande = "AVANCE"
     |
     v
affichage
```

Dans la prochaine leçon, nous remplacerons :

```
affichage
```

par :

```
moteurs
```

39. Dépannage : le client ne trouve pas le serveur

Si :

```
client.connect("EV3_A")
```

échoue, vérifie :

```
1. Le serveur est-il allumé ?

2. Bluetooth est-il activé ?

3. Les deux briques sont-elles associées ?

4. Le nom Bluetooth est-il exactement correct ?

5. Le programme serveur est-il démarré ?

6. Le serveur attend il bien avec wait_for_connection() ?
```

40. Dépannage : le programme reste bloqué

Si le serveur reste sur :

```
Attente du client...
```

ce n’est pas forcément une erreur.

Il exécute :

```
server.wait_for_connection()
```

Il attend simplement un client.

Même principe pour :

```
mailbox.wait()
```

Le programme attend un nouveau message.

41. Dépannage : le mauvais message apparaît

Vérifie le nom de la mailbox.

Serveur :

```
TextMailbox("message", server)
```

Client :

```
TextMailbox("message", client)
```

Les noms doivent correspondre.

Cette configuration est incorrecte :

```
TextMailbox("message", server)
```

et :

```
TextMailbox("commande", client)
```

42. Situation problème

Le serveur utilise :

```
NumericMailbox("speed", server)
```

Le client utilise :

```
TextMailbox("speed", client)
```

Quel est le problème ?

Correction

Les deux programmes n’utilisent pas le même type de mailbox.

Il faut utiliser le même type des deux côtés.

43. Exercice de compréhension

Relie chaque élément à sa fonction.

Élément	Fonction
`BluetoothMailboxServer`	Créer le serveur
`BluetoothMailboxClient`	Créer le client
`TextMailbox`	Transmettre du texte
`NumericMailbox`	Transmettre des nombres
`send()`	Envoyer une valeur
`read()`	Lire une valeur
`wait()`	Attendre un nouveau message

44. Questions de fin

Question 1

Quel EV3 démarre la connexion ?

Question 2

Quel programme doit normalement être lancé en premier ?

Question 3

Quelle instruction permet au serveur d’attendre un client ?

Question 4

Quelle instruction envoie un message ?

Question 5

Quelle instruction lit la mailbox ?

Question 6

Quelle mailbox utiliser pour envoyer du texte ?

Question 7

Quelle mailbox utiliser pour envoyer une vitesse ?

Correction 1

Le client.

Correction 2

Le serveur.

Correction 3

```
server.wait_for_connection()
```

Correction 4

```
mailbox.send(...)
```

Correction 5

```
mailbox.read()
```

Correction 6

`TextMailbox`.

Correction7

`NumericMailbox`.

45. Petit exercice écrit

Complète :

```
Le __________ attend la connexion.

Le __________ démarre la connexion.

La méthode __________ permet d'envoyer une donnée.

La méthode __________ permet de lire une donnée.

Une __________ permet d'envoyer du texte.

Une __________ permet d'envoyer un nombre.
```

Correction

```
Le serveur attend la connexion.

Le client démarre la connexion.

La méthode send() permet d'envoyer une donnée.

La méthode read() permet de lire une donnée.

Une TextMailbox permet d'envoyer du texte.

Une NumericMailbox permet d'envoyer un nombre.
```

46. Erreurs fréquentes

Erreur 1

Le client est lancé avant le serveur.

Solution :

Démarrer le serveur d'abord.

Erreur 2

Mauvais nom Bluetooth.

Solution :

Vérifier exactement le nom de l'EV3 serveur.

Erreur 3

Les deux briques ne sont pas associées.

Solution :

Reprendre la leçon 27.

Erreur 4

Les mailbox n'ont pas le même nom.

Solution :

Utiliser exactement le même nom.

Erreur 5

Les mailbox n'ont pas le même type.

Solution :

Utiliser `TextMailbox` des deux côtés ou `NumericMailbox` des deux côtés.

Erreur 6

Tu penses que `wait()` signifie que le programme a planté.

Solution :

Expliquer que le programme attend volontairement une nouvelle donnée.

Cours 29 : Transformer une brique EV3 en télécommande Bluetooth

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Utiliser les boutons de la brique EV3 comme commandes
2. Envoyer des ordres par Bluetooth
3. Recevoir ces ordres sur un deuxième EV3
4. Commander deux moteurs à distance
5. Créer les commandes `AVANCE`, `RECULE`, `GAUCHE`, `DROITE` et `STOP`
6. Utiliser une boucle pour maintenir la télécommande active
7. Comprendre la différence entre commande locale et commande distante
8. Mettre en place un arrêt de sécurité

Matériel nécessaire

1. Deux briques LEGO Education Mindstorms EV3
2. Deux cartes microSD avec EV3 MicroPython
3. Un robot EV3 équipé de deux moteurs
4. Les moteurs connectés sur les ports B et C
5. Les deux briques déjà associées en Bluetooth
6. Le nom Bluetooth exact de la brique robot

1. Introduction

Lors de la leçon précédente, nous avons appris à envoyer des messages entre deux briques EV3.

Par exemple :

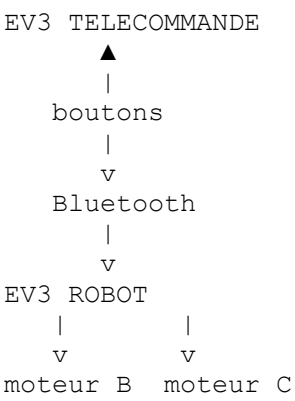
```
EV3 CLIENT

"AVANCE"
     |
     v
Bluetooth
     |
     v
EV3 SERVEUR

"AVANCE"
```

Aujourd'hui, nous allons remplacer le simple affichage du message par une vraie action.

Nous voulons obtenir :



2. Organisation du système

Nous allons utiliser deux briques.

EV3 1

Elle servira de télécommande.

```
EV3 TELECOMMANDE
```

EV3 2

Elle sera installée sur le robot.

```
EV3 ROBOT
```

La télécommande envoie les ordres.

Le robot les reçoit et agit.

3. Qui sera serveur et qui sera client ?

Pour cette leçon, nous allons choisir :

```
EV3 ROBOT = SERVEUR
```

et :

```
EV3 TELECOMMANDE = CLIENT
```

Pourquoi ?

Parce que le robot peut démarrer puis attendre qu'une télécommande se connecte.

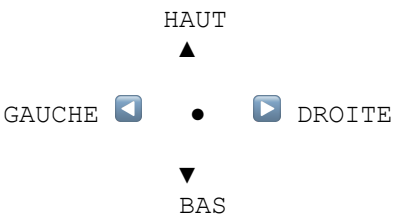
```
ROBOT
SERVEUR

"J'attends"
     ^
     |
 Bluetooth
     |
     |
TELECOMMANDE
CLIENT

"Je me connecte"
```

4. Les boutons de la brique EV3

La brique EV3 possède plusieurs boutons.



Nous allons utiliser cette disposition comme une manette.

Par exemple :

```
HAUT
=
AVANCE
BAS
=
RECULE
GAUCHE
=
TOURNER A GAUCHE
DROITE
=
TOURNER A DROITE
CENTRE
=
STOP
```

5. Lire les boutons avec Pybricks

Nous allons utiliser :

```
from pybricks.hubs import EV3Brick
```

Puis :

```
ev3 = EV3Brick()
```

Les boutons actuellement pressés peuvent être obtenus avec :

```
ev3.buttons.pressed()
```

Cette méthode renvoie une liste des boutons pressés.

6. Importer les boutons

Nous devons également importer :

```
from pybricks.parameters import Button
```

Cela nous permet d'utiliser :

```
Button.UP
Button.DOWN
Button.LEFT
Button.RIGHT
Button.CENTER
```

7. Petit programme de test

Avant d'utiliser Bluetooth, vérifions les boutons.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.parameters import Button
from pybricks.tools import wait

ev3 = EV3Brick()

while True:

    buttons = ev3.buttons.pressed()

    if Button.UP in buttons:
        print("HAUT")

    if Button.DOWN in buttons:
        print("BAS")

    if Button.LEFT in buttons:
        print("GAUCHE")

    if Button.RIGHT in buttons:
        print("DROITE")

    if Button.CENTER in buttons:
        print("CENTRE")

    wait(50)
```

8. Comprendre la boucle

La ligne :

```
while True:
```

permet au programme de fonctionner continuellement.

Sans cette boucle, la brique ne vérifierait les boutons qu'une seule fois.

Avec :

```
while True:
```

le programme fait constamment :

```
Lire boutons
     |
     v
Tester boutons
     |
     v
Lire boutons
     |
     v
Tester boutons
     |
     v
...
```

9. Pourquoi utiliser `wait(50)` ?

La ligne :

```
wait(50)
```

ajoute une petite pause.

Cela évite que la boucle tourne inutilement à une vitesse très élevée.

50 signifie :

```
50 millisecondes
```

soit :

```
0,05 seconde
```

10. Définir nos commandes

Nous allons transmettre des chaînes de caractères.

```
AVANCE
RECULE
GAUCHE
DROITE
STOP
```

Notre protocole de commande sera donc :

Bouton	Message envoyé
Haut	AVANCE
Bas	RECULE
Gauche	GAUCHE
Droite	DROITE
Centre	STOP

11. Programme de la télécommande

Voici une première version.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.parameters import Button
from pybricks.messaging import BluetoothMailboxClient
from pybricks.messaging import TextMailbox
from pybricks.tools import wait

ev3 = EV3Brick()

client = BluetoothMailboxClient()
mailbox = TextMailbox("commande", client)
print("Connexion au robot...")
client.connect("EV3_ROBOT")
print("Connecte")

while True:
    buttons = ev3.buttons.pressed()

    if Button.UP in buttons:
        mailbox.send("AVANCE")

    elif Button.DOWN in buttons:
        mailbox.send("RECULE")

    elif Button.LEFT in buttons:
        mailbox.send("GAUCHE")

    elif Button.RIGHT in buttons:
        mailbox.send("DROITE")

    elif Button.CENTER in buttons:
        mailbox.send("STOP")

    wait(100)
```

Il faut remplacer :

```
EV3_ROBOT
```

par le nom Bluetooth exact de la brique serveur.

12. Pourquoi utiliser elif ?

Nous avons utilisé :

```
if
elif
elif
elif
elif
```

plutôt que plusieurs `if`.

Pourquoi ?

Parce que nous voulons qu'une seule commande principale soit envoyée à chaque passage dans la boucle.

Avec plusieurs `if`, plusieurs conditions pourraient éventuellement être exécutées si plusieurs boutons étaient pressés.

13. Le problème de cette première version

Regarde attentivement.

Si tu maintiens le bouton haut pendant une seconde, le programme peut envoyer :

```
AVANCE
AVANCE
AVANCE
AVANCE
AVANCE
AVANCE
...
```

Ce n'est pas forcément grave, mais c'est inutile.

Nous pouvons améliorer cela plus tard.

Pour commencer, cette version est suffisamment simple.

14. Préparer le robot

Le robot possède deux moteurs.

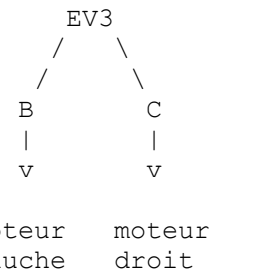
Nous allons utiliser :

```
moteur gauche = port B
```

et :

```
moteur droit = port C
```

Architecture :



15. Importer les moteurs

Nous utilisons :

```
from pybricks.ev3devices import Motor
```

et :

```
from pybricks.parameters import Port
```

Puis :

```
left_motor = Motor(Port.B)
right_motor = Motor(Port.C)
```

16. Programme du robot

Voici la première version complète.

```
#!/usr/bin/env pybricks-micropython

from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.messaging import BluetoothMailboxServer
from pybricks.messaging import TextMailbox

left_motor = Motor(Port.B)
right_motor = Motor(Port.C)

server = BluetoothMailboxServer()

mailbox = TextMailbox("commande", server)

print("Attente telecommande...")

server.wait_for_connection()

print("Telecommande connectee")

while True:

    mailbox.wait()

    commande = mailbox.read()

    print(commande)

    if commande == "AVANCE":

        left_motor.run(500)
        right_motor.run(500)

    elif commande == "RECULE":

        left_motor.run(-500)
        right_motor.run(-500)

    elif commande == "GAUCHE":

        left_motor.run(-300)
        right_motor.run(300)

    elif commande == "DROITE":

        left_motor.run(300)
        right_motor.run(-300)

    elif commande == "STOP":

        left_motor.stop()
        right_motor.stop()
```

17. Comprendre la vitesse des moteurs

L'instruction :

```
left_motor.run(500)
```

fait tourner le moteur à une vitesse donnée.

Une valeur positive :

```
500
```

fait tourner le moteur dans un sens.

Une valeur négative :

```
-500
```

le fait tourner dans l'autre sens.

18. Avancer

Pour avancer, les deux moteurs doivent pousser le robot vers l'avant.

```
left_motor.run(500)
right_motor.run(500)
```

Schéma :

```
GAUCHE         DROITE

  500            500
   ↓              ↓

   O  ---------  O

        AVANCE
          ↑
```

19. Reculer

```
left_motor.run(-500)
right_motor.run(-500)
GAUCHE         DROITE

 -500           -500
   ↓              ↓

   O  ---------  O

        RECULE
          ↓
```

20. Tourner à gauche

Une méthode simple consiste à faire tourner les roues dans des directions opposées.

```
left_motor.run(-300)
right_motor.run(300)
```

Le robot tourne presque sur place.

```
roue gauche
     ↓
   arrière

     ROBOT
      ↺

roue droite
     ↓
   Avant
```

21. Tourner à droite

On inverse :

```
left_motor.run(300)
right_motor.run(-300)
```

Le robot tourne dans l'autre sens.

22. Stop

```
left_motor.stop()
right_motor.stop()
```

Les deux moteurs s'arrêtent.

23. Première manipulation

Procédure :

1. Place le robot au sol
2. Soulève légèrement les roues pour le premier test
3. Démarre le programme serveur sur le robot
4. Attends le message :

```
Attente telecommande...
```

5. Démarre ensuite le programme client
6. Vérifie que les deux briques se connectent
7. Appuie sur les boutons

Teste :

```
HAUT
BAS
GAUCHE
DROITE
CENTRE
```

24. Sécurité

Avant de poser le robot au sol, vérifie que :

```
CENTRE = STOP
```

fonctionne correctement.

C’est important.

Un robot télécommandé doit toujours posséder une commande d’arrêt simple et rapide.

25. Exercice 1

Modifie la vitesse d’avancement.

Passe de :

```
500
```

à :

```
300
```

Observe le comportement.

Puis essaye :

```
700
```

Question :

Quelle vitesse semble la plus facile à contrôler ?

Il n’existe pas forcément une seule bonne réponse.

26. Exercice 2

Modifie la vitesse de rotation.

Actuellement :

```
left_motor.run(-300)
right_motor.run(300)
```

Teste :

```
left_motor.run(-150)
right_motor.run(150)
```

Puis :

```
left_motor.run(-500)
right_motor.run(500)
```

Observe la différence.

27. Exercice 3

Inverse volontairement un moteur pendant l'avance.

```
left_motor.run(500)
right_motor.run(-500)
```

Que fait le robot ?

Réponse

Il tourne au lieu d'avancer.

Cela montre que la direction d'un robot à deux moteurs dépend de la combinaison des vitesses.

28. Le problème du relâchement

Notre télécommande possède actuellement un défaut important.

Supposons que l'on appuie sur :

```
HAUT
```

Le programme envoie :

```
AVANCE
```

Le robot commence à avancer.

Puis l'utilisateur relâche le bouton.

Que se passe t il ?

Avec notre programme actuel :

```
rien
```

Le robot continue d'avancer jusqu'à ce qu'une autre commande soit envoyée.

Ce n'est pas idéal.

29. Ajouter l'arrêt automatique

Nous voulons obtenir :

```
bouton appuyé
=
mouvement
```

et :

```
aucun bouton
=
STOP
```

Nous pouvons modifier le programme télécommande.

```
while True:

    buttons = ev3.buttons.pressed()

    if Button.UP in buttons:
        mailbox.send("AVANCE")

    elif Button.DOWN in buttons:
        mailbox.send("RECULE")

    elif Button.LEFT in buttons:
        mailbox.send("GAUCHE")

    elif Button.RIGHT in buttons:
        mailbox.send("DROITE")

    else:
        mailbox.send("STOP")

    wait(100)
```

Maintenant, lorsqu'aucun bouton n'est pressé :

```
STOP
```

est envoyé.

30. Nouveau problème

Cette amélioration fonctionne, mais elle envoie énormément de messages.

Par exemple si aucun bouton n'est pressé :

```
STOP
STOP
STOP
STOP
STOP
STOP
...
```

Nous allons donc améliorer notre code.

31. Mémoriser la dernière commande

Créons une variable :

```
last_command = None
```

Puis déterminons la commande actuelle.

```
command = "STOP"
```

Ensuite, nous n'envoyons la commande que si elle a changé.

32. Télécommande améliorée

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.parameters import Button
from pybricks.messaging import BluetoothMailboxClient
from pybricks.messaging import TextMailbox
from pybricks.tools import wait

ev3 = EV3Brick()

client = BluetoothMailboxClient()

mailbox = TextMailbox("commande", client)

client.connect("EV3_ROBOT")

last_command = None

while True:

    buttons = ev3.buttons.pressed()

    if Button.UP in buttons:
        command = "AVANCE"

    elif Button.DOWN in buttons:
        command = "RECULE"

    elif Button.LEFT in buttons:
        command = "GAUCHE"

    elif Button.RIGHT in buttons:
        command = "DROITE"

    else:
        command = "STOP"

    if command != last_command:

        mailbox.send(command)

        print(command)

        last_command = command

    wait(50)
```

33. Pourquoi cette version est meilleure ?

Si l'utilisateur maintient le bouton haut :

```
première boucle

AVANCE != STOP
```

Donc :

```
envoi AVANCE
```

Puis :

```
boucle suivante

AVANCE == AVANCE
```

Donc aucun nouvel envoi.

Lorsque le bouton est relâché :

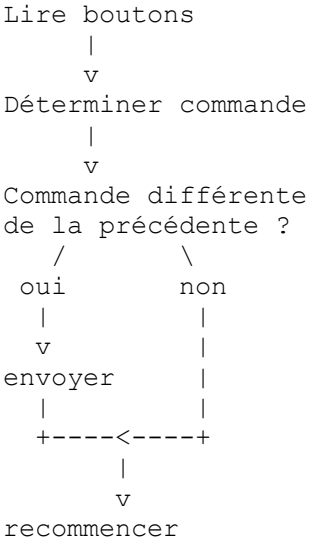
```
STOP != AVANCE
```

Le programme envoie :

```
STOP
```

C'est beaucoup plus propre.

34. Schéma logique



35. Exercice de compréhension

Supposons :

```
last_command = "AVANCE"
```

et :

```
command = "AVANCE"
```

Le programme envoie t il un message ?

Réponse

Non.

Car :

```
command != last_command
```

est faux.

36. Deuxième situation

```
last_command = "AVANCE"
```

et :

```
command = "STOP"
```

Le programme envoie t il un message ?

Réponse

Oui.

Parce que la commande a changé.

37. Mini défi 1

Ajoute une commande spéciale sur le bouton central.

Nous voulons :

```
CENTRE
=
URGENT_STOP
```

Par exemple :

```
if Button.CENTER in buttons:
    command = "URGENT_STOP"
```

Sur le robot :

```
elif commande == "URGENT_STOP":

    left_motor.brake()
    right_motor.brake()
```

38. Différence entre stop() et brake()

Dans Pybricks, plusieurs comportements d'arrêt existent.

Conceptuellement :

```
stop()
=
le moteur n'est plus commandé
```

et :

```
brake()
=
le moteur résiste davantage au mouvement
```

Pour un arrêt de sécurité, `brake()` peut être intéressant.

39. Mini défi 2

Ajouter une vitesse lente.

Par exemple :

```
CENTRE + HAUT
```

pour avancer doucement.

Il faut détecter plusieurs boutons.

Exemple :

```
if Button.CENTER in buttons and Button.UP in buttons:
    command = "AVANCE_LENT"
```

Sur le robot :

```
elif commande == "AVANCE_LENT":

    left_motor.run(200)
    right_motor.run(200)
```

40. Mini défi 3

Créer un mode turbo.

Par exemple :

```
CENTRE + DROITE
```

pour activer :

```
TURBO
```

Puis utiliser :

```
left_motor.run(900)
right_motor.run(900)
```

Il faut évidemment tester prudemment le robot.

41. Utiliser des fonctions

Notre programme robot commence à devenir long.

Nous pouvons l'améliorer avec des fonctions.

```
def avancer():
    left_motor.run(500)
    right_motor.run(500)
def reculer():
    left_motor.run(-500)
    right_motor.run(-500)
def gauche():
    left_motor.run(-300)
    right_motor.run(300)
def droite():
    left_motor.run(300)
    right_motor.run(-300)
def arreter():
    left_motor.stop()
    right_motor.stop()
```

42. Programme robot mieux organisé

```
#!/usr/bin/env pybricks-micropython

from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.messaging import BluetoothMailboxServer
from pybricks.messaging import TextMailbox

left_motor = Motor(Port.B)
right_motor = Motor(Port.C)


def avancer():

    left_motor.run(500)
    right_motor.run(500)


def reculer():

    left_motor.run(-500)
    right_motor.run(-500)


def gauche():

    left_motor.run(-300)
    right_motor.run(300)


def droite():

    left_motor.run(300)
    right_motor.run(-300)


def arreter():

    left_motor.stop()
    right_motor.stop()
```

```
server = BluetoothMailboxServer()

mailbox = TextMailbox("commande", server)

print("Attente telecommande")

server.wait_for_connection()

print("Connecte")

while True:

    mailbox.wait()

    commande = mailbox.read()

    print(commande)

    if commande == "AVANCE":
        avancer()

    elif commande == "RECULE":
        reculer()

    elif commande == "GAUCHE":
        gauche()

    elif commande == "DROITE":
        droite()

    elif commande == "STOP":
        arreter()
```

43. Pourquoi utiliser des fonctions ?

Compare :

```
if commande == "AVANCE":
    left_motor.run(500)
    right_motor.run(500)
```

avec :

```
if commande == "AVANCE":
    avancer()
```

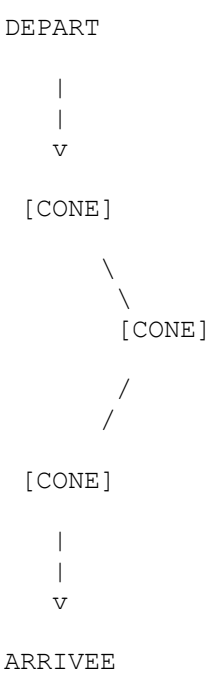
La deuxième version est plus facile à lire.

On comprend immédiatement l'action.

44. Défi final : robot télécommandé

Construis un parcours simple.

Par exemple :



Le robot doit :

1. avancer
2. tourner à gauche
3. tourner à droite
4. reculer
5. s'arrêter précisément
6. ne jamais utiliser de câble entre les deux briques

45. Règles du défi

Tu ne peux toucher que :

```
EV3 TELECOMMANDE
```

Tu ne peux pas toucher physiquement le robot pendant le parcours.

Si le robot quitte complètement la zone, tu dois recommencer.

46. Défi supplémentaire : précision

Créer une zone de stationnement :

```
+------------------+
|                  |
|       ROBOT      |
|                  |
+------------------+
```

Le robot doit s'arrêter complètement dans la zone.

Cela permet de tester la précision de la télécommande.

47. Défi supplémentaire : transmission d'état

Si tu es plus avancé, le robot peut répondre à la télécommande.

Par exemple :

```
TELECOMMANDE

"AVANCE"

      |
      v

ROBOT

      |
      v

"OK"
```

On peut créer une deuxième mailbox :

```
status_box = TextMailbox("status", ...)
```

Le robot pourrait envoyer :

```
AVANCE_OK
STOP_OK
```

Cela introduit une communication bidirectionnelle plus structurée.

48. Notion de protocole

Nous venons en réalité de créer un protocole très simple.

Nous avons décidé ensemble que :

```
AVANCE
```

signifie :

```
les deux moteurs vers l'avant
```

et que :

```
GAUCHE
```

signifie :

```
tourner à gauche
```

Le mot n’a aucun pouvoir magique.

Nous aurions pu utiliser :

```
A
R
G
D
S
```

Par exemple :

```
A = AVANCE
R = RECULE
G = GAUCHE
D = DROITE
S = STOP
```

L’important est que les deux programmes utilisent les mêmes règles.

49. Pourquoi utiliser des commandes courtes ?

Sur un petit réseau, envoyer :

```
A
```

au lieu de :

```
AVANCE
```

peut réduire la quantité de données transmises.
Ici, la différence est minuscule.

Mais cela permet d'introduire une notion importante :

```
PROTOCOLE
```

Les machines doivent se mettre d'accord sur la signification des messages.

50. Situation problème

Le client envoie :

```
FORWARD
```

Mais le serveur vérifie :

```
if commande == "AVANCE":
```

Le robot avance t il ?

Correction

Non.

Les deux programmes n'utilisent pas le même protocole.

51. Dépannage : le robot ne bouge pas

Vérifier dans cet ordre :

```
1. Les moteurs sont ils correctement branchés ?
2. Les ports B et C sont ils corrects ?
3. Le programme serveur est il démarré ?
4. La télécommande est elle connectée ?
5. Le serveur reçoit il les messages ?
6. Les messages correspondent ils aux conditions ?
7. Les moteurs fonctionnent ils sans Bluetooth ?
```

52. Astuce de dépannage

Avant de tester les moteurs, affiche toujours :

```
print(commande)
```

Cela permet de déterminer si le problème vient :

```
du Bluetooth
```

ou :

```
des moteurs
```

Par exemple, si l'écran affiche :

```
AVANCE
```

mais que le robot ne bouge pas, le Bluetooth fonctionne probablement.

Il faut chercher du côté des moteurs.

53. Questions de compréhension

Question 1

Quelle brique joue le rôle de serveur dans notre montage ?

Question 2

Quelle brique joue le rôle de client ?

Question 3

Pourquoi utiliser une boucle infinie ?

Question 4

Quelle méthode permet de connaître les boutons pressés ?

Question 5

Pourquoi envoyer `STOP` quand aucun bouton n'est pressé ?

Question 6

À quoi sert `last_command` ?

Question 7

Pourquoi créer un arrêt de sécurité ?

Correction 1

La brique robot.

Correction 2

La télécommande.

Correction 3

Pour surveiller continuellement les boutons et recevoir continuellement des commandes.

Correction 4

```
ev3.buttons.pressed()
```

Correction 5

Pour éviter que le robot continue son mouvement après le relâchement du bouton.

Correction 6

À éviter d'envoyer continuellement la même commande.

Correction 7

Pour pouvoir immobiliser rapidement le robot en cas de problème.

54. Petit exercice écrit

Complète :

```
Le robot joue le rôle de __________.

La télécommande joue le rôle de __________.

Le bouton UP envoie la commande __________.

La commande STOP permet de __________.

La variable last_command permet d'éviter __________.
```

Correction

```
Le robot joue le rôle de serveur.

La télécommande joue le rôle de client.

Le bouton UP envoie la commande AVANCE.

La commande STOP permet d'arrêter le robot.

La variable last_command permet d'éviter d'envoyer plusieurs fois inutilement la même
commande.
```

55. Erreurs fréquentes

Erreur 1

Le robot continue d'avancer lorsque le bouton est relâché.

Solution :

Envoyer `STOP` lorsqu'aucun bouton n'est pressé.

Erreur 2

Le robot tourne dans le mauvais sens.

Solution :

Vérifier le montage des moteurs et inverser éventuellement le signe d'une vitesse.

Erreur 3

Les commandes sont envoyées continuellement.

Solution :

Utiliser `last_command`.

Erreur 4

Le robot ne reçoit rien.

Solution :

Vérifier le nom de la mailbox et la connexion Bluetooth.

Erreur 5

Le robot reçoit correctement les commandes mais ne bouge pas.

Solution :

Tester les moteurs séparément et vérifier les ports.

Cours 30 : Découvrir les périphériques d'entrée Linux

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Comprendre ce qu'est un périphérique d'entrée
2. Comprendre comment Linux représente les périphériques
3. Identifier le dossier `/dev/input`
4. Comprendre le principe des fichiers `eventX`
5. Utiliser la bibliothèque `evdev`
6. Lister les périphériques d'entrée disponibles
7. Identifier une manette dans la liste
8. Lire les événements générés par un périphérique

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython
3. Un ordinateur
4. Facultativement une manette USB ou Bluetooth compatible
5. Un câble USB si nécessaire pour les premiers tests

1. Introduction

Lors de la leçon précédente, nous avons utilisé les boutons de la brique EV3 comme télécommande.

Nous avions :

```
BOUTON EV3
    |
    v
PROGRAMME
    |
    v
BLUETOOTH
    |
    v
ROBOT
```

Nous voulons maintenant utiliser une vraie manette.

```
MANETTE
   🎮
    |
    v
BLUETOOTH
    |
    v
EV3
    |
    v
PROGRAMME
```

Mais avant de pouvoir utiliser une manette, nous devons comprendre comment Linux la voit.

2. Qu'est-ce qu'un périphérique ?

Un périphérique est un appareil connecté à un ordinateur.

Par exemple :

```
CLAVIER
SOURIS
MANETTE
WEBCAM
IMPRIMANTE
```

Tous ces appareils sont des périphériques.

Certains transmettent des informations vers l'ordinateur.

On les appelle des :

```
PERIPHERIQUES D'ENTREE
```

3. Exemples de périphériques d'entrée

Un clavier transmet :

```
touche A
touche B
touche Enter
```

Une souris transmet :

```
déplacement X
déplacement Y
clic gauche
clic droit
```

Une manette transmet :

```
bouton X
bouton O
joystick gauche
joystick droit
gâchette
```

4. Entrée et sortie

On peut classer les périphériques en deux catégories principales.

Entrée

Ils envoient des informations vers l'ordinateur.

```
MANETTE
   |
   v
ORDINATEUR
```

Exemples :

```
clavier
souris
microphone
manette
```

Sortie

Ils reçoivent des informations venant de l'ordinateur.

```
ORDINATEUR
    |
    v
ECRAN
```

Exemples :

```
écran
haut parleur
imprimante
```

Certains périphériques peuvent faire les deux.

5. Et la manette ?

Une manette est principalement un périphérique d'entrée.

Lorsque l'on appuie sur un bouton :

```
BOUTON X
   |
   v
MANETTE
   |
   v
LINUX
```

Linux reçoit une information.

Cette information est appelée :

```
EVENEMENT
```

6. Qu’est-ce qu’un événement ?

Un événement indique qu’une action vient de se produire.

Par exemple :

```
bouton appuyé
```

ou :

```
bouton relâché
```

ou :

```
joystick déplacé
```

Chaque action produit un événement que notre programme peut lire.

7. Le rôle de Linux

Dans notre EV3, le programme Python ne parle pas directement à la manette.

Le chemin ressemble plutôt à ceci :

```
MANETTE
   |
   v
BLUETOOTH ou USB
   |
   v
LINUX
   |
   v
DRIVER
   |
   v
EVENEMENT
   |
   v
PYTHON
```

8. Qu’est-ce qu’un driver ?

Un driver est un logiciel qui permet au système d’exploitation de comprendre un périphérique.

On peut imaginer le driver comme un traducteur.

```
MANETTE
   |
   v
DRIVER
   |
   v
LINUX
```

La manette envoie ses propres informations.

Le driver les transforme en événements compréhensibles par Linux.

9. Le dossier /dev

Sous Linux, beaucoup de périphériques sont représentés dans le dossier :

```
/dev
```

`dev` vient de :

```
devices
```

c'est à dire :

```
périphériques
```

10. Le dossier /dev/input

Les périphériques d'entrée se trouvent notamment dans :

```
/dev/input
```

Dans ce dossier, Linux peut créer des fichiers ressemblant à :

```
event0
event1
event2
event3
```

Par exemple :

```
/dev/input/event0
/dev/input/event1
/dev/input/event2
```

11. Pourquoi event0, event1, etc. ?

Chaque périphérique d'entrée peut être représenté par un fichier événement.

Par exemple :

```
event0 = boutons EV3
event1 = clavier
event2 = manette
```

Mais attention :

ce numéro n'est pas fixe.

Une manette peut être :

```
event2
```

aujourd'hui et :

```
event3
```

dans une autre configuration.

12. Pourquoi ne pas utiliser directement event2 ?

Supposons que l'on écrive :

```
gamepad = InputDevice("/dev/input/event2")
```

Cela peut fonctionner.

Mais si la manette devient ensuite :

```
/dev/input/event3
```

le programme ne trouvera plus la bonne manette.

Une méthode plus robuste consiste à rechercher le périphérique par son nom.

Nous verrons cela un peu plus loin.

13. La bibliothèque evdev

Pour lire facilement les événements Linux, nous pouvons utiliser une bibliothèque appelée :

```
evdev
```

Elle permet notamment de :

```
chercher les périphériques
ouvrir un périphérique
lire les événements
identifier les boutons
identifier les joysticks
```

14. Premier programme : lister les périphériques

Essayons :

```
from evdev import InputDevice
from evdev import list_devices

devices = list_devices()

for path in devices:
    print(path)
```

15. Résultat possible

La brique peut afficher :

```
/dev/input/event0
/dev/input/event1
/dev/input/event2
```

Pour l'instant, nous connaissons seulement les chemins.

Nous ne savons pas encore à quoi ils correspondent.

16. Afficher le nom du périphérique

Améliorons le programme.

```
from evdev import InputDevice
from evdev import list_devices

devices = list_devices()

for path in devices:

    device = InputDevice(path)

    print(path)
    print(device.name)
```

17. Résultat possible

On pourrait obtenir quelque chose comme :

```
/dev/input/event0
EV3 Buttons

/dev/input/event1
Wireless Controller
```

Le nom exact dépend du périphérique utilisé.

18. Comprendre le programme

Cette instruction :

```
devices = list_devices()
```

demande à Linux :

```
Quels périphériques d'entrée sont disponibles ?
```

Puis :

```
for path in devices:
```

permet d'examiner chaque périphérique.

19. Créer un InputDevice

Cette ligne :

```
device = InputDevice(path)
```

ouvre le périphérique.

Nous pouvons ensuite demander :

```
device.name
```

pour connaître son nom.

20. Premier exercice

Lance le programme sans manette connectée.

Note tous les périphériques affichés.

Par exemple :

```
AVANT CONNEXION

event0
event1
```

Connecte ensuite une manette.

Relance le programme.

Exemple :

```
APRES CONNEXION

event0
event1
event2
```

Quel périphérique est apparu ?

C’est probablement la manette.

21. Méthode de comparaison

Cette technique est très utile pour le dépannage.

```
ETAPE 1

observer avant
ETAPE 2

connecter périphérique
ETAPE 3

observer après
ETAPE 4

Comparer
```

22. Mini défi 1

Écris un programme qui affiche :

```
PERIPHERIQUES DETECTES
```

puis la liste des noms.

Solution possible :

```
from evdev import InputDevice
from evdev import list_devices

print("PERIPHERIQUES DETECTES")

for path in list_devices():

    device = InputDevice(path)

    print(device.name)
```

23. Trouver automatiquement une manette

Supposons que notre manette soit appelée :

```
Wireless Controller
```

Nous pouvons chercher ce nom.

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device
```

24. Vérifier si elle a été trouvée

Ajoutons :

```
if gamepad != None:
    print("Manette detectee")
else:
    print("Manette non detectee")
```

Programme complet :

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

if gamepad != None:
    print("Manette detectee")
else:
    print("Manette non detectee")
```

25. Pourquoi cette méthode est meilleure ?

Nous ne dépendons plus de :

```
event2
```

ou :

```
event3
```

Nous recherchons directement :

```
Wireless Controller
```

Le programme devient plus robuste.

26. Attention au nom exact

Une manette PS3 peut avoir un nom différent d'une manette PS4.

On peut rencontrer des noms tels que :

```
Sony PLAYSTATION(R)3 Controller
```

ou :

```
Wireless Controller
```

Le nom dépend du modèle et du driver utilisé.

Il faut donc toujours commencer par afficher :

```
device.name
```

27. Lire les événements

Une fois le périphérique identifié, nous pouvons observer ce qu'il envoie.

Programme :

```
from evdev import InputDevice

gamepad = InputDevice("/dev/input/event2")

for event in gamepad.read_loop():

    print(event)
```

Il faut évidemment adapter le chemin au périphérique réellement trouvé.

28. Qu’est-ce que read_loop() ?

La méthode :

```
gamepad.read_loop()
```

attend continuellement de nouveaux événements.

On peut représenter son fonctionnement comme ceci :

```
attendre
   |
   v
événement
   |
   v
afficher
   |
   v
attendre
   |
   v
événement
   |
   v
...
```

29. Premier test

Lance le programme.

Ne touche pas à la manette.

Puis appuie sur un bouton.

Tu devrais voir apparaître une ou plusieurs lignes.

Par exemple :

```
event at 1234.56, code 305, type 01, val 1
```

Le résultat exact peut varier.

30. Trois informations importantes

Un événement contient notamment :

```
TYPE
CODE
VALUE
```

Ces trois informations seront fondamentales pour les prochaines leçons.

31. Type

Le type indique la catégorie de l'événement.

Par exemple :

```
bouton
```

ou :

```
axe analogique
```

32. Code

Le code indique quel bouton ou quel axe a généré l'événement.

Exemple :

```
code 305
```

peut correspondre à un bouton particulier selon la manette.

33. Value

La valeur indique l'état ou la position.

Pour un bouton :

```
0
```

peut signifier :

```
relâché
```

et :

```
1
```

peut signifier :

```
appuyé
```

34. Exemple conceptuel

Lorsque l'utilisateur appuie sur X :

```
BOUTON X
   |
   v

type = bouton
code = 305
value = 1
```

Lorsque l'utilisateur relâche X :

```
BOUTON X
   |
   v

type = bouton
code = 305
value = 0
```

35. Et le joystick ?

Un joystick fonctionne différemment.

Il ne possède pas seulement deux états.

Il peut produire de nombreuses valeurs.

Par exemple :

```
0
64
128
192
255
```

Cela permet de connaître la position du joystick.

36. Exemple de joystick

```
GAUCHE         CENTRE        DROITE

  0             128           255

  ◀️----------●----------▶️
```

Les valeurs exactes dépendront de la manette.

Nous les étudierons dans la leçon 33.

37. Activité pratique

Lance :

```
for event in gamepad.read_loop():
    print(event)
```

Puis teste :

1. bouton X
2. bouton carré
3. bouton triangle
4. bouton cercle
5. joystick gauche
6. joystick droit

Note ce qui change.

38. Tableau d’observation

Tu peux remplir le tableau suivant :

Action	Type	Code	Valeur
Bouton X pressé			
Bouton X relâché			
Carré pressé			
Joystick gauche à gauche			
Joystick gauche au centre			
Joystick gauche à droite			

L’objectif n’est pas encore de mémoriser les codes.

Il faut simplement observer.

39. Pourquoi plusieurs événements apparaissent parfois ?

Un seul mouvement de manette peut générer plusieurs événements.

Par exemple :

```
joystick déplacé
```

peut provoquer plusieurs valeurs successives.

```
128
135
150
175
200
```

Le système transmet la progression du mouvement.

40. Défi 2

Crée un programme qui affiche seulement :

```
NOUVEL EVENEMENT
```

à chaque événement reçu.

Solution :

```
from evdev import InputDevice

gamepad = InputDevice("/dev/input/event2")

for event in gamepad.read_loop():

    print("NOUVEL EVENEMENT")
```

41. Défi 3

Affiche uniquement :

```
event.type
event.code
```

et :

```
event.value
```

Programme :

```
from evdev import InputDevice

gamepad = InputDevice("/dev/input/event2")

for event in gamepad.read_loop():

    print("Type :", event.type)
    print("Code :", event.code)
    print("Valeur :", event.value)
```

42. Résultat possible

```
Type : 1
Code : 305
Valeur : 1
```

Puis lors du relâchement :

```
Type : 1
Code : 305
Valeur : 0
```

43. Pourquoi ne pas encore commander un moteur ?

Nous pourrions déjà essayer.

Mais il est préférable de suivre une progression.

Aujourd'hui :

```
MANETTE
   |
   v
EVENEMENT
   |
   v
AFFICHAGE
```

Plus tard :

```
MANETTE
   |
   v
EVENEMENT
   |
   v
CONDITION
   |
   v
MOTEUR
```

Cette méthode permet de vérifier chaque étape séparément.

44. Méthode de dépannage importante

Lorsqu'un système complexe ne fonctionne pas, il faut tester les étapes une par une.

Par exemple :

```
La manette est-elle détectée ?
```

Puis :

```
Les événements apparaissent ils ?
```

Puis :

```
Le bon code est-il détecté ?
```

Puis seulement :

```
Le moteur répond-il ?
```

45. Chaîne complète

```
MANETTE
   |
   v
BLUETOOTH
   |
   v
DRIVER LINUX
   |
   v
/dev/input/eventX
   |
   v
evdev
   |
   v
PROGRAMME PYTHON
```

46. Mini défi 4

Créer un programme qui recherche automatiquement la manette puis lit ses événements.

Exemple :

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    print(device.name)

    if device.name == "Wireless Controller":
        gamepad = device

if gamepad == None:

    print("Manette introuvable")

else:

    print("Manette trouvee")
    print(gamepad.path)

    for event in gamepad.read_loop():

        print(event)
```

47. Pourquoi utiliser None ?

Au départ :

```
gamepad = None
```

signifie :

```
aucune manette trouvée pour le moment
```

Si nous trouvons le bon périphérique :

```
gamepad = device
```

la variable contient maintenant la manette.

48. Situation problème

Le programme contient :

```
if device.name == "Wireless Controller":
```

Mais la liste affiche :

```
Sony PLAYSTATION(R)3 Controller
```

Pourquoi la manette n'est elle jamais trouvée ?

Correction

Parce que le programme recherche un nom différent du nom réel.

Il faut adapter la condition.

49. Améliorer la recherche

Nous pouvons également rechercher une partie du nom.

Par exemple :

```
if "Controller" in device.name:
```

Cela permet de trouver plusieurs périphériques dont le nom contient :

```
Controller
```

Exemple :

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if "Controller" in device.name:
        gamepad = device
```

50. Attention

Cette méthode est plus flexible, mais peut aussi trouver le mauvais périphérique si plusieurs appareils contiennent le mot :

```
Controller
```

Il faut donc utiliser une recherche suffisamment précise.

51. Défi final de la leçon

Tu dois réaliser un programme qui :

1. cherche les périphériques d'entrée
2. affiche leur nom
3. identifie la manette
4. affiche :

```
MANETTE DETECTEE
```

5. affiche son chemin
6. lit tous les événements
7. affiche séparément `type`, `code` et `value`

Exemple de résultat :

```
MANETTE DETECTEE

Nom :
Wireless Controller

Chemin :
/dev/input/event2

Type : 1
Code : 305
Valeur : 1
```

52. Solution possible

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    print("Peripherique :")
    print(device.name)

    if device.name == "Wireless Controller":
        gamepad = device


if gamepad == None:

    print("MANETTE NON DETECTEE")

else:

    print("MANETTE DETECTEE")

    print("Nom :")
    print(gamepad.name)

    print("Chemin :")
    print(gamepad.path)
```

```
for event in gamepad.read_loop():

    print("Type :", event.type)
    print("Code :", event.code)
    print("Valeur :", event.value)
```

53. Questions de compréhension

Question 1

Qu'est ce qu'un périphérique d'entrée ?

Question 2

Donne trois exemples.

Question 3

Dans quel dossier Linux trouve t on principalement les événements d'entrée ?

Question 4

Que représente `event2` ?

Question 5

Pourquoi ne faut il pas supposer que la manette sera toujours `event2` ?

Question 6

Quelle bibliothèque utilisons nous ?

Question 7

Quelles sont les trois informations importantes d'un événement ?

Correction 1

Un appareil qui envoie des informations vers le système.

Correction 2

Clavier, souris, manette.

Correction 3

`/dev/input`

Correction 4

Un périphérique ou une interface d'entrée gérée par Linux.

Correction 5

Parce que le numéro peut changer selon les périphériques connectés et l'ordre de détection.

Correction 6

`evdev`.

Correction 7

Type, code et valeur.

54. Petit exercice écrit

Complète :

```
Une manette est principalement un périphérique d'_____.
Linux représente de nombreux périphériques dans le dossier _____.
Les périphériques d'entrée sont notamment visibles dans _____.
La bibliothèque Python utilisée dans cette leçon s'appelle _____.
Un événement possède notamment un type, un code et une _____.
```

Correction

```
Une manette est principalement un périphérique d'entrée.
Linux représente de nombreux périphériques dans le dossier /dev.
Les périphériques d'entrée sont notamment visibles dans /dev/input.
La bibliothèque Python utilisée dans cette leçon s'appelle evdev.
Un événement possède notamment un type, un code et une valeur.
```

55. Erreurs fréquentes

Erreur 1

Supposer que la manette est toujours `event2`.

Solution :

Lister les périphériques puis chercher par nom.

Erreur 2

La manette n'apparaît pas.

Solution :

Vérifier qu'elle est réellement connectée avant de chercher un problème dans le code.

Erreur 3

Le nom dans le programme ne correspond pas au nom Linux.

Solution :

Afficher d'abord tous les `device.name`.

Erreur 4

Le programme semble bloqué sur `read_loop()`.

Solution :

C'est normal tant qu'aucun événement n'est généré.

Erreur 5

Beaucoup d'événements apparaissent en déplaçant un joystick.

Solution :

C'est normal, car un axe analogique produit de nombreuses valeurs intermédiaires.

Cours 31 : Connecter une manette Bluetooth à la brique EV3

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Préparer la brique EV3 pour une connexion Bluetooth
2. Mettre une manette en mode association
3. Connecter une manette PS3 ou PS4 à l'EV3
4. Comprendre que la procédure dépend du modèle de manette
5. Vérifier que Linux reconnaît correctement la manette
6. Identifier le nom de la manette dans `/dev/input`
7. Retrouver automatiquement son périphérique `eventX`
8. Tester les premiers événements produits par la manette
9. Diagnostiquer les problèmes de connexion les plus fréquents

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython et ev3dev
3. Une manette compatible
4. Idéalement une DualShock 3 PS3 ou une DualShock 4 PS4
5. Pour une DualShock 3, un câble mini USB vers USB A
6. Une batterie de manette suffisamment chargée

1. Introduction

Lors de la leçon précédente, nous avons découvert comment Linux représente une manette.

Nous avons vu cette chaîne :

```
MANETTE
   |
   v
LINUX
   |
   v
/dev/input/eventX
   |
   v
evdev
   |
   v
PROGRAMME PYTHON
```

Mais nous avions supposé que la manette était déjà connectée.

Aujourd'hui, nous allons réellement construire cette connexion :



Une fois la connexion réalisée, nous n'aurons plus besoin d'un ordinateur pour piloter le robot.

Avec une manette PS3 correctement associée, ev3dev indique justement qu'elle peut ensuite se connecter directement à l'EV3 et être utilisée depuis Brickman sans ordinateur.

2. Quelle manette utiliser ?

Pour ce cours, deux modèles sont particulièrement intéressants.

```
DualShock 3
PS3
```

et :

```
DualShock 4
PS4
```

Les deux peuvent fonctionner avec ev3dev, mais leur procédure d'association n'est pas exactement la même.

PS3

La DualShock 3 utilise une procédure particulière.

La première association commence avec un câble USB relié directement à l'EV3.

PS4

La DualShock 4 peut être mise directement en mode association Bluetooth avec les boutons :

```
PS + SHARE
```

puis recherchée depuis le menu Bluetooth de l'EV3.

3. Avant de commencer

Sur la brique EV3, ouvre :

```
Wireless and Networks
```

puis :

```
Bluetooth
```

Vérifie :

```
Powered = ON
```

et :

```
Visible = ON
```

L'EV3 doit donc être :

```
allumé
+
Bluetooth actif
+
Visible
```

4. Pourquoi Powered ?

Si :

```
Powered = OFF
```

le module Bluetooth est désactivé.

Aucune communication ne sera possible.

```
MANETTE )))       EV3

             X

       Bluetooth OFF
```

5. Pourquoi Visible ?

Lorsque :

```
Visible = ON
```

les appareils Bluetooth peuvent découvrir la brique lors de certaines procédures d'association.

C'est particulièrement important pour la procédure PS3.

6. Première méthode : DualShock 3 PS3

Nous allons commencer par la PS3 car sa procédure est particulière.

Le tutoriel ev3dev recommande de respecter précisément l'ordre des opérations.

7. Étape 1 : préparer l'EV3

Sur l'EV3 :

```
Wireless and Networks
        |
        v
Bluetooth
```

Vérifie :

```
Powered
✓

Visible
✓
```

8. Étape 2 : connecter la PS3 en USB

Utilise le port USB A de la brique EV3.

```
PS3
 🎮
  |
  |
mini USB
  |
  |
USB A
  |
  v
 EV3
```

Il faut utiliser un câble permettant réellement le transfert de données.

9. Étape 3 : observer la manette

Dans le menu Bluetooth, un périphérique devrait apparaître sous un nom ressemblant à :

```
PLAYSTATION(R)3 Controller
```

La documentation ev3dev précise qu'à cette étape, il ne faut pas lancer une association Bluetooth manuelle depuis ce périphérique.

10. Étape 4 : retirer le câble

Débranche maintenant le câble USB.

```
PS3          EV3
```



```
plus de câble
```

11. Étape 5 : appuyer sur PS

Appuie ensuite sur :

```
PS
```

au centre de la manette.

La manette tente alors de se connecter à la brique.

12. Étape 6 : autoriser le service HID

L'EV3 peut afficher une demande ressemblant à :

```
Authorize service HID?
```

Choisis :

```
Accept
```

HID signifie :

```
Human Interface Device
```

c'est à dire un périphérique d'interface humaine.

Par exemple :

```
clavier
souris
manette
```

Cette demande d'autorisation HID fait partie de la procédure PS3 documentée par ev3dev.

13. Résultat attendu PS3

La manette est maintenant associée.

Lors des utilisations suivantes, il suffit normalement d'appuyer sur :

```
PS
```

pour qu'elle tente de se reconnecter à la brique EV3 associée.

Nous obtenons :

```
PS3
 🎮

  )))
  )))
Bluetooth
  )))
  )))

 EV3
```

14. Deuxième méthode : DualShock 4 PS4

La procédure PS4 est différente.

Nous n'avons normalement pas besoin de commencer avec un câble USB.

15. Mettre la PS4 en mode association

Sur la DualShock 4, maintiens simultanément :

```
PS
+
SHARE
```

pendant quelques secondes.

La manette passe en mode association Bluetooth. La procédure ev3dev pour le robot PS4Explor3r utilise précisément cette combinaison.

16. Rechercher depuis l'EV3

Sur la brique :

```
Wireless and Networks
        |
        v
Bluetooth
        |
        v
Start Scan
```

La brique recherche les appareils disponibles.

Après quelques instants, nous devrions voir :

```
Wireless Controller
```

17. Sélectionner la manette

Choisis :

```
Wireless Controller
```

puis effectue l'association.

Le projet PS4Explor3r d'ev3dev utilise précisément cette méthode pour associer une DualShock 4 à l'EV3.

18. Résultat attendu PS4

```
DUALSHOCK 4

      )))
      )))
   Bluetooth
      )))
      )))

      EV3
```

La manette est maintenant prête à être reconnue comme périphérique d'entrée Linux.

19. Pourquoi les procédures sont-elles différentes ?

Bluetooth décrit la technologie de communication.

Mais chaque périphérique peut avoir ses propres mécanismes d'association.

Ainsi :

```
PS3
```

et :

```
PS4
```

utilisent toutes les deux Bluetooth, mais la procédure initiale n'est pas identique.

Il ne faut donc pas supposer que toutes les manettes Bluetooth se configurent exactement de la même façon.

20. Vérifier la connexion

Une fois la manette associée, nous allons reprendre ce que nous avons appris dans la leçon 30.

Programme :

```
from evdev import InputDevice
from evdev import list_devices

for path in list_devices():

    device = InputDevice(path)

    print(device.path)
    print(device.name)
```

21. Résultat possible avec une PS3

On peut obtenir un nom similaire à :

```
PLAYSTATION(R)3 Controller
```

La documentation ev3dev utilise d'ailleurs ce nom pour identifier la manette dans son exemple Python.

22. Résultat possible avec une PS4

On peut obtenir :

```
Wireless Controller
```

Le chemin pourrait être :

```
/dev/input/event2
```

ou :

```
/dev/input/event3
```

Le numéro n'est pas garanti.

23. Pourquoi vérifier le nom ?

Nous ne voulons jamais supposer :

```
PS4 = event2
```

Nous voulons plutôt demander à Linux :

```
Quel périphérique s'appelle
Wireless Controller ?
```

24. Programme pour une PS4

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

if gamepad == None:
    print("Manette non detectee")

else:
    print("Manette detectee")
    print(gamepad.name)
    print(gamepad.path)
```

25. Programme pour une PS3

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device

if gamepad == None:
    print("Manette non detectee")

else:
    print("Manette detectee")
    print(gamepad.name)
    print(gamepad.path)
```

26. Programme compatible avec plusieurs manettes

Nous pouvons améliorer le programme.

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    print(device.name)

    if device.name == "Wireless Controller":
        gamepad = device

    elif device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device

if gamepad == None:

    print("Aucune manette compatible")

else:

    print("Manette detectee")
    print(gamepad.name)
    print(gamepad.path)
```

27. Pourquoi ce programme est intéressant ?

Il permet au même programme de fonctionner avec :

```
PS3
```

ou :

```
PS4
```

sans devoir connaître :

```
event1
event2
event3
```

à l'avance.

28. Premier test d’événements

Une fois `gamepad` trouvé :

```
for event in gamepad.read_loop():

    print(event)
```

Programme complet :

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

    elif device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device


if gamepad == None:

    print("Manette non detectee")

else:

    print("Manette connectee")

    for event in gamepad.read_loop():

        print(event)
```

29. Manipulation

Appuie successivement sur :

`X`

puis :

`Carré`

puis :

`Triangle`

puis :

`Cercle`

Enfin, déplace les deux joysticks.

L’objectif est uniquement de vérifier que des événements apparaissent.

30. Test de connexion réussi

Si nous obtenons :

```
Manette connectee
```

puis des événements lorsque nous appuyons sur les boutons :

```
Bluetooth fonctionne
        +
Linux voit la manette
        +
evdev reçoit les événements
```

La chaîne complète fonctionne.

31. Architecture obtenue

```
BOUTON PHYSIQUE
       |
       v
     MANETTE
       |
       v
   Bluetooth
       |
       v
      EV3
       |
       v
     Linux
       |
       v
/dev/input/eventX
       |
       v
     evdev
       |
       v
     Python
```

32. Activité 1

Déconnecte la manette.
Lance le programme de recherche.

Résultat attendu :

```
Manette non detectee
```

Reconnecte ensuite la manette.
Relance le programme.

Résultat attendu :

```
Manette detectee
```

33. Activité 2

Note dans un tableau les informations obtenues.

Information	Valeur
Modèle	PS3 ou PS4
Nom Linux	
Chemin	
Connexion	Bluetooth
Événements reçus	Oui ou non

34. Activité 3

Éteins complètement la manette.

Puis rallume la avec :

```
PS
```

Observe si elle se reconnecte automatiquement à l'EV3.

Pour une PS3 correctement configurée selon la procédure ev3dev, le bouton PS lui permet de tenter de se reconnecter à l'EV3 associé.

35. Mini défi 1

Créer un programme qui affiche seulement :

```
PS3 DETECTEE
```

ou :

```
PS4 DETECTEE
```

selon la manette.

Solution possible :

```
from evdev import InputDevice
from evdev import list_devices

for path in list_devices():

    device = InputDevice(path)

    if device.name == "PLAYSTATION(R)3 Controller":
        print("PS3 DETECTEE")

    elif device.name == "Wireless Controller":
        print("PS4 DETECTEE")
```

36. Mini défi 2

Créer trois variables :

```
gamepad = None
model = None
path_found = None
```

Puis faire afficher :

```
Modele : PS4

Peripherique :
/dev/input/event2
```

ou l'équivalent correspondant à la manette utilisée.

37. Mini défi 3

Lorsque la manette est détectée, faire produire un son à l'EV3.

Nous pouvons utiliser Pybricks et evdev dans le même programme.

```
from pybricks.hubs import EV3Brick
from evdev import InputDevice
from evdev import list_devices

ev3 = EV3Brick()

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

    elif device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device


if gamepad != None:

    print("Manette detectee")

    ev3.speaker.beep()
```

Nous combinons maintenant :

```
LINUX
+
evdev
+
PYBRICKS
```

dans le même programme.

38. Pourquoi est-ce important ?

C’est exactement ce dont nous avons besoin pour notre futur robot.

```
evdev
```

va lire :

```
la manette
```

et :

```
Pybricks
```

va contrôler :

```
les moteurs
les LEDs
l'écran
le haut parleur
les capteurs
```

39. Dépannage : la PS3 n’apparaît pas

Vérifie :

```
1. Bluetooth Powered est activé

2. Visible est activé

3. La manette est chargée

4. Le câble USB transporte bien les données

5. La manette apparaît lorsqu'elle est branchée

6. Le câble a ensuite été retiré

7. Le bouton PS a été pressé

8. L'autorisation HID a été acceptée
```

40. Dépannage : la PS4 n’apparaît pas

Vérifie :

```
1. Bluetooth est actif

2. PS et Share ont été maintenus

3. La manette est réellement en mode association

4. Start Scan a été lancé

5. Wireless Controller apparaît

6. La manette n'est pas déjà connectée à un autre appareil proche
```

41. Dépannage : la manette apparaît dans Bluetooth mais pas dans Python

Il faut distinguer :

`ASSOCIEE`

de :

`CONNECTEE`

et :

`VISIBLE COMME PERIPHERIQUE D'ENTREE`

Une association Bluetooth réussie ne signifie pas nécessairement que `/dev/input/eventX` existe à cet instant.

Vérifie avec :

```
for path in list_devices():
    print(InputDevice(path).name)
```

42. Dépannage : le programme cherche le mauvais nom

Supposons :

```
if device.name == "Wireless Controller":
```

mais Linux affiche :

`PLAYSTATION(R)3 Controller`

La condition ne sera jamais vraie.

Toujours afficher les noms réels avant de programmer la détection.

43. Dépannage : event2 a changé

Hier :

```
/dev/input/event2
```

Aujourd'hui :

```
/dev/input/event3
```

Ce n'est pas nécessairement un problème.

Linux a simplement attribué un autre numéro.

C'est pourquoi nous cherchons la manette par :

```
device.name
```

et non uniquement par numéro.

44. Situation problème

Si tu écris :

```
gamepad = InputDevice("/dev/input/event2")
```

Cela fonctionne.

Tu redémarres la brique et le programme ne fonctionne plus.

La manette est maintenant :

```
/dev/input/event3
```

Quelle amélioration faut-il apporter ?

Correction

Rechercher automatiquement le périphérique avec `list_devices()` et son nom.

45. Situation problème PS3

Tu branches la manette PS3 à l'EV3.

Tu vois :

```
PLAYSTATION(R)3 Controller
```

puis clique immédiatement sur une option d'association Bluetooth.

Pourquoi peut ce être un problème ?

Correction

Parce que la procédure PS3 ev3dev demande de ne pas associer la manette à cette étape. Il faut la détecter par USB, retirer le câble, appuyer sur PS puis accepter l'autorisation HID.

46. Situation problème PS4

Tu lances :

```
Start Scan
```

mais la DualShock 4 n'apparaît pas.

Quelle étape as tu peut être oubliée ?

Correction

Mettre la DualShock 4 en mode association en maintenant :

```
PS + SHARE
```

47. Défi final de la leçon

Créer un programme appelé par exemple :

```
gamepad_test.py
```

qui doit :

1. rechercher les périphériques Linux
2. détecter une PS3 ou une PS4
3. identifier le modèle
4. afficher le nom Linux
5. afficher le chemin `eventX`
6. produire un bip lorsque la manette est trouvée
7. commencer à lire les événements
8. afficher `type`, `code` et `value`

48. Solution possible

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick

from evdev import InputDevice
from evdev import list_devices


ev3 = EV3Brick()

gamepad = None
model = None


print("Recherche de la manette...")


for path in list_devices():

    device = InputDevice(path)

    print(device.name)

    if device.name == "PLAYSTATION(R)3 Controller":

        gamepad = device
        model = "PS3"

    elif device.name == "Wireless Controller":

        gamepad = device
        model = "PS4"


if gamepad == None:

    print("MANETTE NON DETECTEE")

else:

    print("MANETTE DETECTEE")

    print("Modele :")
    print(model)

    print("Nom :")
    print(gamepad.name)

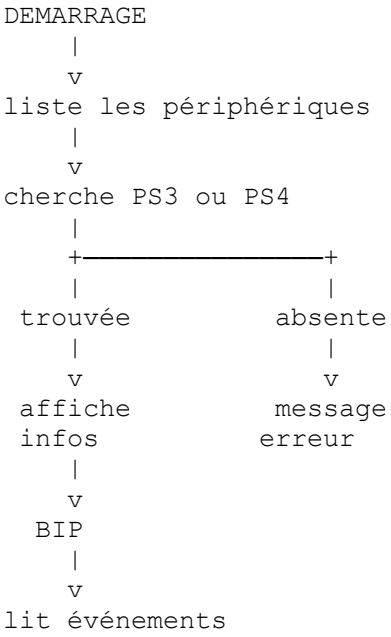
    print("Chemin :")
    print(gamepad.path)

    ev3.speaker.beep()

    for event in gamepad.read_loop():

        print("Type :", event.type)
        print("Code :", event.code)
        print("Valeur :", event.value)
```

49. Ce programme fait quoi ?



50. Questions de compréhension

Question 1

Quelle technologie utilisons nous pour connecter la manette sans fil ?

Question 2

Comment commence l'association d'une DualShock 3 avec cette procédure ev3dev ?

Question 3

Quels boutons permettent de mettre une DualShock 4 en mode association ?

Question 4

Quel nom peut avoir une DualShock 4 dans Linux ?

Question 5

Pourquoi rechercher la manette par nom ?

Question 6

Quelle bibliothèque permet de lire les événements ?

Question 7

Avons nous encore besoin d'un ordinateur lorsque la manette est correctement connectée au robot ?

Correction 1

Bluetooth.

Correction 2

En reliant d'abord la manette à l'EV3 avec un câble USB de données.

Correction 3

`PS + SHARE`

Correction 4

`Wireless Controller`

Correction 5

Parce que son numéro `eventX` peut changer.

Correction 6

`evdev`.

Correction 7

Non, pas pour utiliser le robot. Le programme s'exécute directement sur l'EV3 et la manette communique directement avec lui.

51. Petit exercice écrit

Complète :

```
La manette communique avec l'EV3 grâce au __________.

Une manette PS4 peut apparaître sous le nom __________.

Les périphériques d'entrée Linux apparaissent notamment dans __________.

Le numéro eventX peut __________.

La bibliothèque utilisée pour lire les événements est __________.
```

Correction

```
La manette communique avec l'EV3 grâce au Bluetooth.

Une manette PS4 peut apparaître sous le nom Wireless Controller.

Les périphériques d'entrée Linux apparaissent notamment dans /dev/input.

Le numéro eventX peut changer.

La bibliothèque utilisée pour lire les événements est evdev.
```

52. Ce qu’il faut retenir

Pour PS3 :

```
Bluetooth ON
      |
      v
Visible ON
      |
      v
PS3 branchée en USB
      |
      v
détection
      |
      v
retirer USB
      |
      v
bouton PS
      |
      v
accepter HID
      |
      v
CONNECTEE
```

Pour PS4 :

```
PS + SHARE
      |
      v
mode association
      |
      v
Start Scan EV3
      |
      v
Wireless Controller
      |
      v
association
      |
      v
CONNECTEE
```

Puis dans les deux cas :

```
MANETTE
    |
    v
/dev/input/eventX
    |
    v
evdev
    |
    v
PYTHON
```

53. Erreurs fréquentes

Erreur 1

Confondre association et connexion.

Erreur 2

Utiliser la procédure PS4 avec une PS3.

Erreur 3

Utiliser un câble USB qui recharge mais ne transporte pas les données.

Erreur 4

Coder directement `/dev/input/event2`.

Erreur 5

Ne pas vérifier le nom réel de la manette.

Erreur 6

Chercher un problème dans Python alors que la manette n’est pas réellement connectée.

Cours 32 : Lire les boutons d'une manette Bluetooth

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Lire les événements d'une manette
2. Différencier un bouton appuyé d'un bouton relâché
3. Comprendre les notions `type`, `code` et `value`
4. Identifier les codes des principaux boutons
5. Créer une table de correspondance entre codes et boutons
6. Réagir à un bouton précis
7. Commander un moteur à partir d'un bouton
8. Créer plusieurs actions différentes avec la manette

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython
3. Une manette PS3 ou PS4 déjà connectée
4. Un robot EV3
5. Un ou deux moteurs
6. La bibliothèque `evdev` disponible dans l'environnement

1. Introduction

Lors de la leçon précédente, nous avons réussi à connecter une manette à la brique EV3.

Nous pouvions obtenir des événements comme :

```
Type : 1
Code : 305
Valeur : 1
```

Mais ce résultat n'est pas très parlant.

Nous voulons maintenant transformer :

```
305
```

en :

```
BOUTON X
```

Puis transformer :

```
BOUTON X
```

en :

```
FAIRE AVANCER LE ROBOT
```

Notre chaîne devient donc :

```
MANETTE
   |
   v
EVENEMENT
   |
   v
IDENTIFICATION DU BOUTON
   |
   v
ACTION
```

2. Rappel : structure d’un événement

Un événement contient plusieurs informations.

Les trois principales sont :

```
type
code
value
```

Exemple :

```
type = 1
code = 305
value = 1
```

3. Le type

Le champ :

```
event.type
```

indique la catégorie d'événement.

Pour un bouton, on rencontre généralement :

```
type = 1
```

Sous Linux, cela correspond aux événements de type bouton ou touche.

4. Le code

Le champ :

```
event.code
```

indique quel bouton a généré l'événement.

Par exemple, sur une DualShock 4, un code peut correspondre à :

```
X
Triangle
Carré
Cercle
L1
R1
```

Les valeurs exactes peuvent dépendre du modèle et du pilote.

Il faut donc toujours les vérifier expérimentalement.

5. La valeur

Le champ :

```
event.value
```

indique l'état du bouton.

Pour un bouton classique :

```
0 = relâché
1 = appuyé
```

Nous pouvons donc avoir :

```
code = 305
value = 1
```

qui signifie :

```
bouton 305 appuyé
```

Puis :

```
code = 305
value = 0
```

qui signifie :

```
bouton 305 relâché
```

6. Première expérience

Utilise ce programme :

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

    elif device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device


if gamepad == None:

    print("Manette non detectee")

else:

    print("Manette detectee")

    for event in gamepad.read_loop():

        print("Type :", event.type)
        print("Code :", event.code)
        print("Valeur :", event.value)
```

7. Activité d’observation

Appuie uniquement sur :

```
X
```

Note le code obtenu.

Puis fais de même pour :

```
Carré
Triangle
Cercle
L1
R1
```

Remplis ton propre tableau.

Bouton	Code observé
X	
Carré	
Triangle	
Cercle	
L1	
R1	

8. Pourquoi faire notre propre tableau ?

Parce que le code peut varier selon :

```
le modèle de manette
le pilote
la version du système
```

Il est donc plus fiable de vérifier réellement les événements.

9. Exemple avec une DualShock 4

Sur certaines configurations, on peut observer par exemple :

Bouton	Code
Carré	304
X	305
Cercle	306
Triangle	307
L1	308
R1	309

Nous utiliserons ces valeurs comme exemples.

Mais tu dois vérifier sa propre manette.

10. Détecter uniquement les boutons

Nous pouvons ignorer les autres événements.

Par exemple :

```
for event in gamepad.read_loop():

    if event.type == 1:

        print("Code :", event.code)
        print("Valeur :", event.value)
```

Nous nous concentrons maintenant uniquement sur les boutons.

11. Détecter un appui

Nous voulons afficher un message uniquement au moment où un bouton est appuyé.

```
if event.type == 1 and event.value == 1:
    print("Bouton appuye")
```

Ainsi, le relâchement n'est pas affiché.

12. Détecter le bouton X

Supposons que :

```
X = 305
```

Nous pouvons écrire :

```
if event.type == 1:

    if event.code == 305 and event.value == 1:
        print("X appuye")
```

13. Tester plusieurs boutons

```
if event.type == 1 and event.value == 1:

    if event.code == 305:
        print("X")

    elif event.code == 304:
        print("CARRE")

    elif event.code == 307:
        print("TRIANGLE")

    elif event.code == 306:
        print("CERCLE")
```

14. Résultat attendu

Lorsque tu appuies sur X :

```
X
```

Sur Carré :

```
CARRE
```

Sur Triangle :

```
TRIANGLE
```

15. Créer des constantes

Écrire directement :

```
if event.code == 305:
```

fonctionne, mais ce n'est pas très lisible.

Nous pouvons écrire :

```
BUTTON_X = 305
BUTTON_SQUARE = 304
BUTTON_TRIANGLE = 307
BUTTON_CIRCLE = 306
```

Puis :

```
if event.code == BUTTON_X:
    print("X")
```

C'est beaucoup plus clair.

16. Programme mieux organisé

```
BUTTON_SQUARE = 304
BUTTON_X = 305
BUTTON_CIRCLE = 306
BUTTON_TRIANGLE = 307

for event in gamepad.read_loop():
    if event.type == 1 and event.value == 1:

        if event.code == BUTTON_X:
            print("X")

        elif event.code == BUTTON_SQUARE:
            print("CARRE")

        elif event.code == BUTTON_CIRCLE:
            print("CERCLE")

        elif event.code == BUTTON_TRIANGLE:
            print("TRIANGLE")
```

17. Pourquoi utiliser des constantes ?

Compare :

```
if event.code == 305:
```

avec :

```
if event.code == BUTTON_X:
```

La deuxième version est beaucoup plus facile à comprendre.

18. Utiliser un dictionnaire

On peut également utiliser un dictionnaire Python.

```
buttons = {
    304: "CARRE",
    305: "X",
    306: "CERCLE",
    307: "TRIANGLE"
}
```

Puis :

```
if event.code in buttons:

    print(buttons[event.code])
```

19. Programme avec dictionnaire

```
buttons = {
    304: "CARRE",
    305: "X",
    306: "CERCLE",
    307: "TRIANGLE",
    308: "L1",
    309: "R1"
}

for event in gamepad.read_loop():

    if event.type == 1 and event.value == 1:

        if event.code in buttons:

            print(buttons[event.code])
```

20. Avantage du dictionnaire

Nous évitons :

```
if
elif
elif
elif
elif
...
```

Le programme devient plus compact.

21. Première commande de moteur

Nous allons maintenant connecter un moteur sur le port B.

Importations :

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
```

Création :

```
motor = Motor(Port.B)
```

22. Bouton X pour démarrer

Nous voulons :

```
X
|
v
MOTEUR TOURNE
```

Programme :

```
if event.code == BUTTON_X and event.value == 1:

    motor.run(500)
```

23. Carré pour arrêter

```
if event.code == BUTTON_SQUARE and event.value == 1:

    motor.stop()
```

Nous obtenons :

```
X
↓
RUN

CARRE
↓
STOP
```

24. Programme complet simple

```
#!/usr/bin/env pybricks-micropython

from pybricks.ev3devices import Motor
from pybricks.parameters import Port

from evdev import InputDevice
from evdev import list_devices


BUTTON_SQUARE = 304
BUTTON_X = 305

motor = Motor(Port.B)

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

if gamepad == None:

    print("Manette non detectee")

else:

    print("Manette connectee")

    for event in gamepad.read_loop():

        if event.type == 1:

            if event.code == BUTTON_X and event.value == 1:

                print("Moteur ON")
                motor.run(500)

            elif event.code == BUTTON_SQUARE and event.value == 1:

                print("Moteur STOP")
                motor.stop()
```

25. Activité 1

Modifie le programme pour obtenir :

```
X
=
moteur avant
Cercle
=
moteur arrière
Carré
=
Stop
```

26. Solution possible

```
if event.type == 1 and event.value == 1:

    if event.code == BUTTON_X:
        motor.run(500)

    elif event.code == BUTTON_CIRCLE:
        motor.run(-500)

    elif event.code == BUTTON_SQUARE:
        motor.stop()
```

27. Utiliser le relâchement

Nous pouvons aussi décider que le moteur ne tourne que tant que X est maintenu.

Appui :

```
value = 1
```

Relâchement :

```
value = 0
```

Nous pouvons donc écrire :

```
if event.code == BUTTON_X:

    if event.value == 1:
        motor.run(500)

    elif event.value == 0:
        motor.stop()
```

28. Pourquoi cette version est intéressante ?

Le comportement devient :

```
maintenir X
=
moteur tourne
```

puis :

```
relâcher X
=
moteur stop
```

C’est plus proche du fonctionnement d’une vraie télécommande.

29. Exercice

Créer :

```
maintenir X
=
avancer
```

et :

```
maintenir Cercle
=
reculer
```

Lorsque le bouton est relâché :

```
STOP
```

30. Attention avec plusieurs boutons

Supposons que X soit maintenu.

Le moteur avance.

Puis tu appuies sur Cercle.

Nous pouvons obtenir des comportements inattendus si notre programme ne gère pas bien l’état des boutons.

Pour nos premiers exercices, nous allons demander de n’utiliser qu’un bouton à la fois.

31. Commander deux moteurs

Connectons maintenant :

```
moteur gauche = B
moteur droit = C
```

Programme :

```
left_motor = Motor(Port.B)
right_motor = Motor(Port.C)
```

32. Créer des fonctions

```
def avancer():

    left_motor.run(500)
    right_motor.run(500)


def reculer():

    left_motor.run(-500)
    right_motor.run(-500)


def arreter():

    left_motor.stop()
    right_motor.stop()
```

33. Associer les boutons

Nous pouvons décider :

```
X
=
AVANCE
Cercle
=
RECULE
Carré
=
STOP
Triangle
=
KLAXON
```

34. Ajouter le haut parleur

```
from pybricks.hubs import EV3Brick

ev3 = EV3Brick()
```

Puis :

```
ev3.speaker.beep()
```

35. Exemple complet

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

from evdev import InputDevice
from evdev import list_devices


BUTTON_SQUARE = 304
BUTTON_X = 305
BUTTON_CIRCLE = 306
BUTTON_TRIANGLE = 307


ev3 = EV3Brick()

left_motor = Motor(Port.B)
right_motor = Motor(Port.C)


def avancer():

    left_motor.run(500)
    right_motor.run(500)


def reculer():

    left_motor.run(-500)
    right_motor.run(-500)


def arreter():

    left_motor.stop()
    right_motor.stop()


gamepad = None


for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":

        gamepad = device


if gamepad == None:

    print("Manette non detectee")

else:

    print("Manette connectee")

    for event in gamepad.read_loop():

        if event.type == 1 and event.value == 1:
```

```
            if event.code == BUTTON_X:

                print("AVANCE")
                avancer()

            elif event.code == BUTTON_CIRCLE:

                print("RECULE")
                reculer()

            elif event.code == BUTTON_SQUARE:

                print("STOP")
                arreter()

            elif event.code == BUTTON_TRIANGLE:

                print("KLAXON")
                ev3.speaker.beep()
```

36. Ajouter L1 et R1

Supposons :

```
L1 = 308
R1 = 309
```

Nous pouvons les utiliser pour tourner.

```
BUTTON_L1 = 308
BUTTON_R1 = 309
```

Puis :

```
elif event.code == BUTTON_L1:

    left_motor.run(-300)
    right_motor.run(300)
```

et :

```
elif event.code == BUTTON_R1:

    left_motor.run(300)
    right_motor.run(-300)
```

37. Nouveau tableau de commandes

Bouton	Action
X	Avancer
Cercle	Reculer
Carré	Stop
Triangle	Klaxon
L1	Tourner à gauche
R1	Tourner à droite

38. Mini défi 1

Créer les six commandes du tableau.

Le robot doit pouvoir :

```
avancer
reculer
tourner à gauche
tourner à droite
stopper
klaxonner
```

39. Mini défi 2

Faire produire deux sons différents.

Par exemple :

```
Triangle
=
bip court
```

et :

```
R1
=
bip différent
```

On peut utiliser différentes fréquences ou durées.

40. Mini défi 3

Changer la couleur des voyants EV3.

Par exemple :

```
Triangle
=
LED verte
```

et :

```
Carré
=
LED rouge
```

Selon les fonctions disponibles dans la version Pybricks utilisée.

41. Défi supplémentaire : afficher l’action

À chaque commande, affiche :

```
AVANCE
```

ou :

```
RECULE
```

ou :

```
GAUCHE
```

sur l’EV3.

Cela permet de voir rapidement ce que le programme comprend.

42. Défi supplémentaire : arrêt lors du relâchement

Modifier le programme pour que :

```
X appuyé
=
avance
```

et :

```
X relâché
=
stop
```

Même chose pour Cercle, L1 et R1.

43. Exemple

```
if event.code == BUTTON_X:

    if event.value == 1:

        avancer()

    elif event.value == 0:

        arreter()
```

44. Pourquoi cette logique est plus sûre ?

Si la commande dépend uniquement de l'appui :

```
X
```

le robot peut continuer à avancer après le relâchement.

Avec la deuxième méthode :

```
bouton maintenu
=
mouvement

bouton relâché
=
stop
```

le contrôle devient plus naturel.

45. Situation problème

Le programme contient :

```
if event.code == 305:
    avancer()
```

Que manque t il ?

Correction

Le programme ne vérifie pas si le bouton vient d'être appuyé ou relâché.

Il faudrait par exemple tester :

```
event.value == 1
```

46. Situation problème

Tu appuies sur X.

Le programme affiche :

```
X
```

deux fois.

Pourquoi ?

Correction

Un événement peut être généré lors de l'appui et un autre lors du relâchement.

Il faut regarder `event.value`.

47. Situation problème

Le code indique :

```
BUTTON_X = 305
```

Mais X ne déclenche rien.

Que faut il faire en premier ?

Correction

Afficher les événements réels de la manette et vérifier le code associé à X.

48. Méthode de diagnostic

Toujours suivre :

```
1. La manette est-elle connectée ?
2. Les événements apparaissent ils ?
3. Le type est-il correct ?
4. Le code est-il celui attendu ?
5. La valeur change t-elle ?
6. La condition Python est-elle correcte ?
7. L'action fonctionne t-elle séparément ?
```

49. Défi final : télécommande par boutons

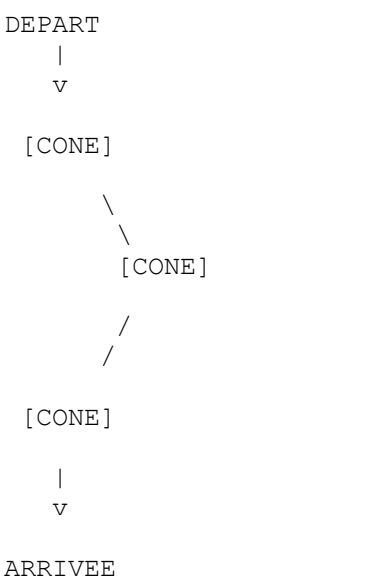
Créer un robot pouvant être contrôlé avec la manette.

Cahier des charges :

```
X
=
AVANCE
CERCLE
=
RECULE
L1
=
GAUCHE
R1
=
DROITE
CARRE
=
STOP
TRIANGLE
=
KLAXON
```

50. Parcours pratique

Créer un petit parcours :



Tu dois réaliser le parcours uniquement avec la manette.

51. Limitation de cette méthode

Notre contrôle fonctionne avec des boutons.

Mais il reste assez brutal.

Par exemple :

```
X
=
500
```

Le robot passe immédiatement de :

```
0
```

à :

```
500
```

Il n’y a pas de vitesse intermédiaire.

52. Ce que nous voulons

Avec un joystick :

```
petit mouvement
=
petite vitesse
grand mouvement
=
grande vitesse
```

Par exemple :

```
joystick à 25 %
=
moteur à 25 %
```

et :

```
joystick à 100 %
=
moteur à 100 %
```

53. Cela nécessite les axes analogiques

Les boutons produisent essentiellement :

```
0
ou
1
```

Les joysticks produisent beaucoup plus de valeurs.

Par exemple :

```
0
32
64
96
128
160
192
224
255
```

Nous allons donc quitter les événements de type bouton et observer les événements analogiques.

54. Questions de compréhension

Question 1

Que représente `event.code` ?

Question 2

Que représente généralement `event.value = 1` pour un bouton ?

Question 3

Que représente généralement `event.value = 0` ?

Question 4

Pourquoi utiliser des constantes comme `BUTTON_X` ?

Question 5

Pourquoi faut il vérifier les codes sur sa propre manette ?

Question 6

Quel avantage offre le relâchement d'un bouton ?

Correction 1

Le bouton ou l'axe qui a généré l'événement.

Correction 2

Le bouton est appuyé.

Correction 3

Le bouton est relâché.

Correction 4

Pour rendre le programme plus lisible.

Correction 5

Ils peuvent varier selon le modèle et le pilote.

Correction 6

Il peut être utilisé pour arrêter immédiatement le mouvement.

55. Petit exercice écrit

Complète :

```
event.type indique le __________ d'événement.

event.code permet d'identifier le __________.

event.value indique notamment si le bouton est __________ ou __________.

Une valeur 1 signifie généralement __________.

Une valeur 0 signifie généralement __________.
```

Correction

```
event.type indique le type d'événement.

event.code permet d'identifier le bouton.

event.value indique notamment si le bouton est appuyé ou relâché.

Une valeur 1 signifie généralement appuyé.

Une valeur 0 signifie généralement relâché.
```

56. Erreurs fréquentes

Erreur 1

Les codes ne correspondent pas à la manette.

Solution :

Refaire le tableau d'observation.

Erreur 2

Le robot continue à avancer.

Solution :

Utiliser l'événement de relâchement ou un bouton Stop.

Erreur 3

Le programme réagit aussi au relâchement.

Solution :

Tester :

```
event.value == 1
```

Erreur 4

Le programme ne trouve plus la manette.

Solution :

Reprendre la détection par nom de la leçon 31.

Erreur 5

Les moteurs fonctionnent mais dans le mauvais sens.

Solution :

Adapter le signe des vitesses selon le montage.

Cours 33 : Utiliser les joysticks analogiques d'une manette

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Comprendre la différence entre un bouton et un axe analogique
2. Lire la valeur d'un joystick
3. Identifier les axes horizontal et vertical
4. Comprendre la notion de valeur centrale
5. Convertir une valeur de joystick en vitesse moteur
6. Créer une zone morte
7. Limiter une valeur
8. Contrôler progressivement la vitesse d'un robot
9. Utiliser un joystick pour avancer, reculer et tourner

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython
3. Une manette PS3 ou PS4 déjà connectée
4. Un robot EV3 avec deux moteurs
5. Les moteurs branchés sur les ports B et C
6. La bibliothèque `evdev`

1. Introduction

Lors de la leçon précédente, nous avons utilisé les boutons de la manette.

Un bouton fonctionne essentiellement avec deux états :

```
0 = relâché

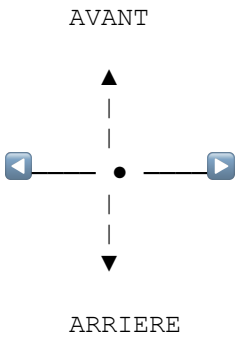
1 = appuyé
```

Cela convient très bien pour :

```
STOP
KLAXON
MODE TURBO
```

Mais pour piloter précisément un robot, nous voulons quelque chose de plus progressif.

C'est le rôle des joysticks analogiques.



2. Bouton ou joystick

Un bouton fournit généralement :

```
0
ou
1
```

Un joystick fournit de nombreuses valeurs.

Par exemple :

```
0
32
64
96
128
160
192
224
255
```

Cela permet de savoir non seulement dans quelle direction le joystick est déplacé, mais aussi de combien.

3. Pourquoi dit-on analogique ?

Un bouton fonctionne presque comme un interrupteur.

```
OFF
ON
```

Un joystick peut prendre de nombreuses positions intermédiaires.

Par exemple :

```
0 %    25 %    50 %    75 %    100 %
```

Nous pouvons donc utiliser l'amplitude du mouvement du joystick pour commander l'amplitude du mouvement du robot.

4. Exemple

Petit mouvement du joystick :

```
Joystick
   ↑
20 %
```

peut produire :

```
Moteur
20 %
```

Grand mouvement :

```
Joystick
   ↑
100 %
```

peut produire :

```
Moteur
100 %
```

5. Les deux axes d’un joystick

Un joystick possède généralement deux axes.

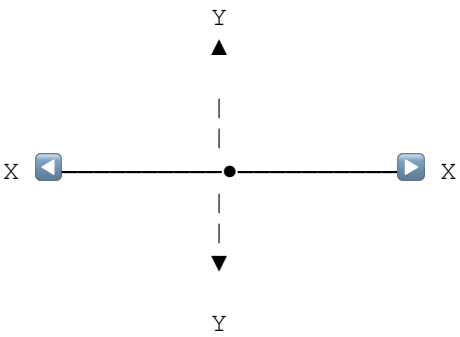
L’axe horizontal :

```
X
```

et l’axe vertical :

```
Y
```

Schéma :



6. Axe X

L'axe X représente généralement :

```
GAUCHE
CENTRE
DROITE
```

7. Axe Y

L'axe Y représente généralement :

```
AVANT
CENTRE
ARRIERE
```

Attention toutefois : selon la manette, pousser vers le haut peut produire une valeur plus petite, et pousser vers le bas une valeur plus grande.

Il faudra donc observer les vraies valeurs avant de programmer.

8. Trouver les événements analogiques

Nous allons reprendre notre programme d'observation.

```
from evdev import InputDevice
from evdev import list_devices

gamepad = None

for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

    elif device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device


if gamepad == None:

    print("Manette non detectee")

else:

    for event in gamepad.read_loop():

        print("Type :", event.type)
        print("Code :", event.code)
        print("Valeur :", event.value)
```

9. Observer uniquement les joysticks

Lorsque tu déplaces un joystick, tu devrais observer des événements différents de ceux des boutons.

Sous Linux, les axes analogiques sont généralement de type :

```
EV_ABS
```

ou numériquement :

```
type = 3
```

Nous pouvons donc filtrer :

```
if event.type == 3:

    print("Code :", event.code)
    print("Valeur :", event.value)
```

10. Activité d’observation

Ne touche d’abord qu’au joystick gauche.

Effectue :

```
gauche
droite
haut
bas
centre
```

et remplis un tableau.

Mouvement	Code	Valeur
Gauche		
Centre horizontal		
Droite		
Haut		
Centre vertical		
Bas		

11. Pourquoi cette étape est essentielle

Comme pour les boutons, les codes peuvent varier selon :

```
le modèle de manette
le pilote
la configuration Linux
```

Il est donc préférable de découvrir les axes expérimentalement.

12. Exemple de plage

Supposons que l'axe vertical donne :

```
0 à 255
```

avec :

```
128 = centre
```

Nous obtenons :

```
HAUT         CENTRE         BAS

 0             128           255
```

Selon le modèle, le sens peut être inversé.

13. Pourquoi la valeur centrale est importante

Lorsque personne ne touche au joystick, nous voulons :

```
vitesse = 0
```

Mais la manette renvoie peut être :

```
128
```

Il faut donc transformer :

```
128
```

en :

```
0
```

14. Première conversion

Supposons :

```
centre = 128
```

Nous pouvons simplement écrire :

```
value = event.value

speed = value - 128
```

Exemples :

```
value = 128

speed = 0
value = 255

speed = 127
value = 0

speed = 128
```

Le dernier résultat est négatif :

```
0 - 128 = 128
```

15. Nouvelle plage

Nous avons maintenant environ :

```
128 à +127
```

avec :

```
0 = centre
```

C'est déjà beaucoup plus pratique.

16. Mais la vitesse n'est pas encore adaptée

Pybricks commande généralement les moteurs en degrés par seconde.

Nous pouvons par exemple vouloir :

```
800 à +800
```

au lieu de :

```
128 à +127
```

Il faut donc effectuer une mise à l'échelle.

17. Mise à l'échelle

Nous pouvons écrire :

```
speed = (value - 128) * 800 / 127
```

Puis :

```
speed = int(speed)
```

18. Exemples

Si :

```
value = 128
```

alors :

```
speed ≈ 0
```

Si :

```
value = 255
```

alors :

```
speed ≈ 800
```

Si :

```
value = 0
```

alors :

```
speed ≈ 806
```

On pourra ensuite limiter précisément la valeur.

19. Inverser le sens

Il est fréquent qu'en poussant le joystick vers le haut, le robot recule.

Pourquoi ?

Parce que la valeur de l'axe Y peut diminuer lorsque le joystick monte.

Nous pouvons simplement inverser le signe :

```
speed = (128 - value) * 800 / 127
```

Ainsi :

```
joystick vers le haut
=
vitesse positive
```

20. Premier test avec un moteur

Connecte un moteur sur le port B.

```
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

motor = Motor(Port.B)
```

Supposons que le code de l'axe Y gauche soit :

```
1
```

Exemple :

```
if event.type == 3 and event.code == 1:

    value = event.value

    speed = int((128 - value) * 800 / 127)

    motor.run(speed)
```

21. Résultat attendu

Joystick au centre :

```
moteur presque arrêté
```

Joystick légèrement en avant :

```
moteur lent
```

Joystick complètement en avant :

```
moteur rapide
```

Joystick en arrière :

```
rotation inverse
```

22. Problème : le moteur bouge tout seul

Même lorsqu'un joystick semble parfaitement centré, il peut renvoyer :

```
126
```

ou :

```
130
```

au lieu de :

```
128
```

Cela peut produire une petite vitesse.

Le robot se met alors à avancer lentement sans que personne ne touche à la manette.

23. La zone morte

Pour corriger ce problème, nous allons créer une :

```
ZONE MORTE
```

ou :

```
DEAD ZONE
```

Le principe :

```
petites valeurs proches de zéro
=
Ignorées
```

24. Exemple

Après conversion :

```
7
```

devient :

```
0
5
```

devient :

```
0
```

mais :

```
80
```

reste :

```
80
```

25. Code

```
if speed > -80 and speed < 80:

    speed = 0
```

Nous pouvons également écrire :

```
if abs(speed) < 80:

    speed = 0
```

26. Comprendre abs()

`abs()` donne la valeur absolue.

Exemples :

```
abs(50) = 50
abs(50) = 50
```

Ainsi :

```
if abs(speed) < 80:
```

fonctionne aussi bien pour les petites valeurs positives que négatives.

27. Fonction de conversion

Créons une fonction.

```
def convert_axis(value):

    speed = int((128 - value) * 800 / 127)

    if abs(speed) < 80:
        speed = 0

    return speed
```

28. Pourquoi créer une fonction ?

Parce que nous aurons plusieurs axes.

Par exemple :

```
joystick gauche Y
joystick droit X
```

Au lieu de répéter le calcul, nous utilisons :

```
convert_axis(value)
```

29. Limiter les valeurs

Notre conversion peut parfois produire légèrement plus que la valeur prévue.

Créons :

```
def limit(value, minimum, maximum):

    if value > maximum:
        return maximum

    if value < minimum:
        return minimum

    return value
```

Puis :

```
speed = limit(speed, -800, 800)
```

30. Fonction complète

```
def convert_axis(value):

    speed = int((128 - value) * 800 / 127)

    if abs(speed) < 80:
        speed = 0

    if speed > 800:
        speed = 800

    if speed < -800:
        speed = -800

    return speed
```

31. Exercice 1

Affiche :

```
valeur brute
```

et :

```
vitesse calculée
```

Exemple :

```
print("Brut :", value)
print("Vitesse :", speed)
```

Déplace lentement le joystick et observe.

32. Exercice 2

Teste différentes zones mortes.

```
20
50
80
150
```

Question :

Quelle zone morte offre le meilleur compromis entre stabilité et précision ?

33. Contrôler deux moteurs

Nous voulons maintenant commander notre robot.

```
left_motor = Motor(Port.B)
right_motor = Motor(Port.C)
```

La première méthode consiste à utiliser l'axe Y pour faire avancer les deux moteurs.

```
left_motor.run(speed)
right_motor.run(speed)
```

34. Résultat

Joystick vertical :

```
AVANT
   |
   v
moteur gauche +
moteur droit +
```

Joystick en arrière :

```
ARRIERE
   |
   v
moteur gauche
moteur droit
```

35. Ajouter la direction

Nous avons besoin d'un deuxième axe.

Par exemple :

```
axe Y gauche
=
accélération
```

et :

```
axe X gauche
=
direction
```

Nous allons stocker deux variables.

```
throttle = 0
steering = 0
```

36. Comprendre throttle

`throttle` représente :

```
avancer
ou
reculer
```

Par exemple :

```
+600 = avancer

0 = stop

600 = reculer
```

37. Comprendre steering

`steering` représente :

```
tourner à gauche
ou
tourner à droite
```

Par exemple :

```
400 = droite

0 = tout droit

400 = gauche
```

Le signe peut être inversé selon la configuration choisie.

38. Mélanger accélération et direction

Nous pouvons utiliser :

```
left_speed = throttle + steering
right_speed = throttle - steering
```

39. Exemple 1 : tout droit

```
throttle = 500
steering = 0
```

Donc :

```
left = 500

right = 500
```

Le robot avance tout droit.

40. Exemple 2 : tourner à droite

```
throttle = 500
steering = 200
```

Alors :

```
left = 700

right = 300
```

La roue gauche tourne plus vite.

Le robot tourne progressivement.

41. Exemple 3 : tourner sur place

```
throttle = 0
steering = 400
```

Résultat :

```
left = 400

right = 400
```

Les roues tournent dans des sens opposés.

Le robot pivote.

42. Limiter après le mélange

Supposons :

```
throttle = 700
steering = 400
```

Alors :

```
left = 1100
```

Nous voulons limiter à :

```
800
```

Il faut donc faire :

```
left_speed = limit(left_speed, -800, 800)
right_speed = limit(right_speed, -800, 800)
```

43. Programme de principe

Les codes d'axes ci dessous sont uniquement des exemples.

Il faut les remplacer par ceux observés avec la manette utilisée.

```
AXIS_LEFT_X = 0
AXIS_LEFT_Y = 1

throttle = 0
steering = 0

for event in gamepad.read_loop():

    if event.type == 3:

        if event.code == AXIS_LEFT_Y:

            throttle = convert_axis(event.value)

        elif event.code == AXIS_LEFT_X:

            steering = convert_axis(event.value)

        left_speed = throttle + steering
        right_speed = throttle - steering

        left_speed = limit(left_speed, -800, 800)
        right_speed = limit(right_speed, -800, 800)

        left_motor.run(left_speed)
        right_motor.run(right_speed)
```

44. Attention au centre

Si :

```
throttle = 0
```

et :

```
steering = 0
```

nous voulons arrêter les moteurs.

On peut utiliser :

```
if left_speed == 0 and right_speed == 0:

    left_motor.stop()
    right_motor.stop()
```

sinon :

```
left_motor.run(left_speed)
right_motor.run(right_speed)
```

45. Fonction de commande du robot

Créons :

```
def drive_robot(throttle, steering):

    left_speed = throttle + steering
    right_speed = throttle - steering

    left_speed = limit(left_speed, -800, 800)
    right_speed = limit(right_speed, -800, 800)

    if left_speed == 0 and right_speed == 0:

        left_motor.stop()
        right_motor.stop()

    else:

        left_motor.run(left_speed)
        right_motor.run(right_speed)
```

46. Organisation du programme

Nous avons maintenant trois niveaux.

```
NIVEAU 1

Lire joystick
NIVEAU 2

Convertir valeur
NIVEAU 3

Commander moteurs
```

Architecture :

```
JOYSTICK
    |
    v
event.value
    |
    v
convert_axis()
    |
    v
throttle / steering
    |
    v
drive_robot()
    |
    v
MOTEURS
```

47. Programme complet d’exemple

Attention : les codes d’axes et la plage `0 à 255` doivent être vérifiés sur la manette utilisée.

```
#!/usr/bin/env pybricks-micropython

from pybricks.ev3devices import Motor
from pybricks.parameters import Port

from evdev import InputDevice
from evdev import list_devices


AXIS_LEFT_X = 0
AXIS_LEFT_Y = 1

MAX_SPEED = 800
DEAD_ZONE = 80


left_motor = Motor(Port.B)
right_motor = Motor(Port.C)


def limit(value, minimum, maximum):

    if value > maximum:
        return maximum
```

```
    if value < minimum:
        return minimum

    return value


def convert_axis(value):

    speed = int((128 - value) * MAX_SPEED / 127)

    if abs(speed) < DEAD_ZONE:
        speed = 0

    return limit(speed, -MAX_SPEED, MAX_SPEED)


def drive_robot(throttle, steering):

    left_speed = throttle + steering
    right_speed = throttle - steering

    left_speed = limit(left_speed, -MAX_SPEED, MAX_SPEED)
    right_speed = limit(right_speed, -MAX_SPEED, MAX_SPEED)

    if left_speed == 0 and right_speed == 0:

        left_motor.stop()
        right_motor.stop()

    else:

        left_motor.run(left_speed)
        right_motor.run(right_speed)


gamepad = None


for path in list_devices():

    device = InputDevice(path)

    if device.name == "Wireless Controller":
        gamepad = device

    elif device.name == "PLAYSTATION(R)3 Controller":
        gamepad = device


if gamepad == None:

    print("Manette non detectee")

else:

    print("Manette connectee")

    throttle = 0
    steering = 0

    for event in gamepad.read_loop():

        if event.type == 3:

            if event.code == AXIS_LEFT_Y:

                throttle = convert_axis(event.value)
```

```
        elif event.code == AXIS_LEFT_X:
            steering = convert_axis(event.value)
        drive_robot(throttle, steering)
```

48. Première amélioration : direction moins sensible

Actuellement, accélération et direction peuvent avoir la même amplitude.

Cela peut rendre le robot très nerveux.

Nous pouvons réduire la direction.

Par exemple :

```
steering = int(convert_axis(event.value) * 0.6)
```

Ainsi :

```
accélération maximale = 800
direction maximale ≈ 480
```

49. Exercice 3

Teste :

```
0.3
0.5
0.7
1.0
```

comme coefficient de direction.

Observe lequel permet le contrôle le plus agréable.

50. Deuxième méthode : deux joysticks

On peut également utiliser :

```
joystick gauche
=
moteur gauche
```

et :

```
joystick droit
=
moteur droit
```

Cette méthode ressemble au pilotage d'un char.

51. Tank drive

Architecture :

```
JOYSTICK GAUCHE
       |
       v
MOTEUR GAUCHE


JOYSTICK DROIT
       |
       v
MOTEUR DROIT
```

Chaque joystick commande directement une roue.

52. Exemple

```
left_speed = convert_axis(left_y)
right_speed = convert_axis(right_y)
```

Puis :

```
left_motor.run(left_speed)
right_motor.run(right_speed)
```

53. Avantages du tank drive

Il permet :

```
un contrôle très direct
des rotations sur place
une logique simple
```

54. Inconvénient

Il peut être plus difficile à maîtriser pour un débutant.

C’est pourquoi nous commencerons généralement avec :

```
un joystick
=
accélération + direction
```

55. Mini défi 1

Contrôler un seul moteur avec un joystick.

Conditions :

```
centre = stop
avant = rotation positive
arrière = rotation négative
vitesse proportionnelle
```

56. Mini défi 2

Contrôler deux moteurs avec uniquement l'axe Y.

Le robot doit pouvoir :

```
avancer progressivement
```

et :

```
reculer progressivement
```

57. Mini défi 3

Ajouter l'axe X.

Le robot doit pouvoir :

```
avancer
reculer
tourner progressivement
pivoter
```

58. Mini défi 4

Ajouter une zone morte réglable.

Créer :

```
DEAD_ZONE = 80
```

Puis tester plusieurs valeurs.

Tu dois expliquer pourquoi une valeur trop grande réduit la précision du joystick.

59. Défi supplémentaire : mode lent

Utilisons un bouton de la manette.

Par exemple :

```
L1 maintenu
=
mode lent
```

En mode normal :

```
MAX_SPEED = 800
```

En mode lent :

```
MAX_SPEED = 300
```

Cela permettra de faire des manœuvres précises.

60. Défi supplémentaire : mode turbo

Par exemple :

```
R1
=
turbo
```

On peut choisir :

```
normal = 600
turbo = 900
```

Il faut évidemment adapter les valeurs au robot et tester avec prudence.

61. Défi supplémentaire : afficher la vitesse

Afficher :

```
Throttle : 430
Steering : 120
```

Cela permet d'observer les valeurs en temps réel.

62. Défi final : parcours analogique

Construire un parcours comportant :

```
une ligne droite
un virage large
un virage serré
une marche arrière
une zone de stationnement
```

Le robot doit réaliser le parcours uniquement avec le joystick analogique.

63. Critère important

Tu ne dois pas uniquement réussir le parcours.

Tu dois démontrer que tu peux faire varier la vitesse.

Par exemple :

```
ligne droite
=
rapide
virage
=
moyen
stationnement
=
très lent
```

64. Situation problème

Le joystick est au centre.

La valeur reçue est :

```
131
```

Le robot avance légèrement.

Que faut il ajouter ?

Correction

Une zone morte.

65. Situation problème

Le robot recule lorsque le joystick est poussé vers l'avant.

Que peut on faire ?

Correction

Inverser la formule ou le signe.

Par exemple passer de :

```
value - 128
```

à :

```
128 - value
```

66. Situation problème

En tournant et avançant simultanément :

```
left_speed = 1100
```

alors que la vitesse maximale choisie est :

```
800
```

Que faut il faire ?

Correction

Limiter les valeurs après le mélange avec une fonction comme `limit()`.

67. Situation problème

Le robot réagit énormément au moindre mouvement horizontal.

Quelle amélioration peut-on faire ?

Correction

Réduire la sensibilité de direction.

Par exemple :

```
steering = int(steering * 0.5)
```

68. Questions de compréhension

Question 1

Quelle est la différence principale entre un bouton et un joystick ?

Question 2

Quels sont les deux axes d’un joystick ?

Question 3

À quoi sert la valeur centrale ?

Question 4

Pourquoi convertir la valeur du joystick ?

Question 5

Qu’est-ce qu’une zone morte ?

Question 6

Pourquoi limiter les vitesses après avoir mélangé accélération et direction ?

Correction 1

Un bouton possède principalement deux états, tandis qu'un joystick fournit de nombreuses valeurs intermédiaires.

Correction 2

X et Y.

Correction 3

Elle représente normalement la position de repos du joystick.

Correction 4

Pour obtenir une plage adaptée à la commande des moteurs.

Correction 5

Une plage autour du centre dans laquelle les petites variations du joystick sont ignorées.

Correction 6

Parce que la somme peut dépasser la vitesse maximale autorisée.

69. Petit exercice écrit

Complète :

```
Un joystick possède généralement un axe ______ et un axe ______.

La position de repos est appelée position ______.

Une zone morte permet d'ignorer les petites ______ autour du centre.

La variable throttle représente principalement ______ et ______.

La variable steering représente la ______.
```

Correction

```
Un joystick possède généralement un axe X et un axe Y.

La position de repos est appelée position centrale.

Une zone morte permet d'ignorer les petites variations autour du centre.

La variable throttle représente principalement l'avance et le recul.

La variable steering représente la direction.
```

74. Erreurs fréquentes

Erreur 1

Utiliser les codes d’axes d’une autre manette.

Solution :

Observer les événements.

Erreur 2

Supposer que le centre vaut toujours exactement 128.

Solution :

Mesurer la valeur réelle.

Erreur 3

Oublier la zone morte.

Solution :

Ajouter un seuil autour de zéro.

Erreur 4

Le sens vertical est inversé.

Solution :

Inverser le calcul.

Erreur 5

Les valeurs dépassent la vitesse maximale.

Solution :

Utiliser `limit()`.

Erreur 6

Le robot est trop difficile à contrôler.

Solution :

Réduire `MAX_SPEED` et la sensibilité de `steering`.

Cours 34 : Construire un robot EV3 radiocommandé par manette Bluetooth

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Regrouper les notions vues dans les leçons précédentes
2. Détecter automatiquement une manette
3. Lire boutons et joysticks
4. Piloter deux moteurs de manière proportionnelle
5. Ajouter un mode lent
6. Ajouter un mode rapide
7. Ajouter un klaxon
8. Ajouter un arrêt d'urgence
9. Organiser le programme avec des fonctions
10. Créer un robot radiocommandé complet et utilisable

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython
3. Une manette PS3 ou PS4 déjà associée
4. Un robot à deux roues
5. Deux grands moteurs EV3
6. Moteur gauche sur le port B
7. Moteur droit sur le port C
8. Facultativement un capteur ultrasonique pour le bonus sécurité

1. Introduction

Nous avons maintenant toutes les briques nécessaires.

Nous savons utiliser :

`Bluetooth`

`evdev`

`boutons`

`joysticks`

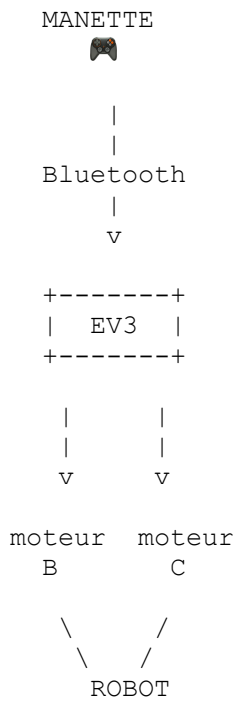
`moteurs`

`fonctions`

`conditions`

`variables`

Nous allons tout regrouper dans un seul système.



2. Cahier des charges

Notre robot doit pouvoir :

```
avancer
reculer
tourner progressivement
pivoter
s'arrêter
```

Il doit également posséder :

```
une zone morte

un mode lent

un mode normal

un mode rapide

un klaxon

un arrêt d'urgence
```

3. Organisation des commandes

Nous allons utiliser cette configuration.

Commande	Action
Joystick gauche Y	Avancer et reculer
Joystick gauche X	Tourner
Carré	Arrêt d'urgence
Triangle	Klaxon
L1	Mode lent
R1	Mode rapide

Le mode normal sera utilisé lorsque ni L1 ni R1 ne sont maintenus.

4. Architecture du programme

Nous allons séparer notre programme en plusieurs blocs.

```
1. Imports

2. Constantes

3. Initialisation

4. Fonctions

5. Recherche de la manette

6. Lecture des événements

7. Commande des moteurs
```

Cette organisation est beaucoup plus propre qu'un seul gros bloc de code.

5. Les imports

Nous avons besoin de Pybricks pour contrôler l'EV3.

```
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
```

Et d'evdev pour lire la manette.

```
from evdev import InputDevice
from evdev import list_devices
```

6. Initialisation

```
ev3 = EV3Brick()

left_motor = Motor(Port.B)
right_motor = Motor(Port.C)
```

7. Les constantes

Nous allons éviter d'écrire des nombres partout.

Par exemple :

```
MODE_SLOW = 300
MODE_NORMAL = 600
MODE_FAST = 900
```

Et :

```
DEAD_ZONE = 80
```

Pour les boutons :

```
BUTTON_SQUARE = 304
BUTTON_TRIANGLE = 307
BUTTON_L1 = 308
BUTTON_R1 = 309
```

Attention : ces codes sont des exemples fréquents pour certaines configurations PS4. Il faut conserver les codes réellement trouvés pendant le **cours 32**.

8. Les axes

Nous utiliserons par exemple :

```
AXIS_LEFT_X = 0
AXIS_LEFT_Y = 1
```

Là encore, les valeurs doivent être adaptées aux observations faites pendant la leçon 33.

9. Fonction de limitation

Créons :

```
def limit(value, minimum, maximum):

    if value > maximum:
        return maximum

    if value < minimum:
        return minimum

    return value
```

Cette fonction garantit que nos commandes ne dépassent jamais les limites choisies.

10. Fonction de conversion du joystick

```
def convert_axis(value, max_speed):

    speed = int((128 - value) * max_speed / 127)

    if abs(speed) < DEAD_ZONE:
        speed = 0

    return limit(speed, -max_speed, max_speed)
```

Cette fonction effectue quatre choses.

```
centrage
mise à l'échelle
zone morte
limitation
```

11. Première remarque importante

Dans cette fonction :

```
DEAD_ZONE = 80
```

est exprimé dans la même unité que la vitesse calculée.

Si le mode lent utilise seulement :

```
300
```

alors une dead zone de 80 représente une part relativement importante du joystick.

Une meilleure solution consiste à appliquer la zone morte avant la mise à l’échelle.

12. Version améliorée

```
def convert_axis(value, max_speed):

    centered = 128 - value

    if abs(centered) < 12:
        return 0

    speed = int(centered * max_speed / 127)

    return limit(speed, -max_speed, max_speed)
```

Nous utilisons maintenant :

```
DEAD_ZONE_RAW = 12
```

Ce seuil agit directement sur la valeur brute de la manette.

13. Pourquoi est-ce mieux ?

La zone morte reste identique quel que soit le mode.

```
MODE LENT
300

MODE NORMAL
600

MODE RAPIDE
900
```

Le joystick possède toujours la même petite zone centrale ignorée.

14. Fonction de pilotage

Nous allons créer :

```
def drive_robot(throttle, steering, max_speed):

    left_speed = throttle + steering
    right_speed = throttle - steering

    left_speed = limit(left_speed, -max_speed, max_speed)
    right_speed = limit(right_speed, -max_speed, max_speed)

    if left_speed == 0 and right_speed == 0:

        left_motor.stop()
        right_motor.stop()

    else:

        left_motor.run(left_speed)
        right_motor.run(right_speed)
```

15. Mélange direction et accélération

Le principe est :

```
gauche = accélération + direction

droite = accélération - direction
```

Exemple :

```
throttle = 400
steering = 0
```

donne :

```
gauche = 400
droite = 400
```

Le robot avance.

16. Exemple de virage

```
throttle = 400
steering = 150
```

donne :

```
gauche = 550
droite = 250
```

Le robot tourne.

17. Rotation sur place

```
throttle = 0
steering = 300
```

donne :

```
gauche = 300
droite = 300
```

avec un moteur dans le sens opposé selon notre formule et notre orientation physique.

Le robot pivote.

18. Les modes de vitesse

Créons :

```
speed_mode = MODE_NORMAL
```

Au démarrage :

```
MODE NORMAL
```

est actif.

19. Bouton L1

Lorsque L1 est appuyé :

```
speed_mode = MODE_SLOW
```

Nous obtenons :

```
MODE LENT
```

Ce mode est utile pour :

```
stationnement
passage étroit
petite correction
```

20. Bouton R1

Lorsque R1 est appuyé :

```
speed_mode = MODE_FAST
```

Nous obtenons :

```
MODE RAPIDE
```

Ce mode est adapté aux lignes droites.

21. Retour au mode normal

Lors du relâchement de L1 ou R1, nous pouvons revenir au mode normal.

```
if event.code == BUTTON_L1 and event.value == 0:
    speed_mode = MODE_NORMAL
```

Même chose pour R1.

22. Klaxon

Triangle peut produire un son.

```
if event.code == BUTTON_TRIANGLE and event.value == 1:

    ev3.speaker.beep()
```

23. Arrêt d’urgence

Carré sera notre bouton le plus important.

```
if event.code == BUTTON_SQUARE and event.value == 1:

    left_motor.brake()
    right_motor.brake()
```

Nous allons également mémoriser :

```
emergency_stop = True
```

Pourquoi ?

Parce que sans cela, un événement joystick juste après l’arrêt pourrait faire repartir immédiatement le robot.

24. Verrouillage de sécurité

Lorsque Carré est appuyé :

```
emergency_stop = True
```

Puis :

```
left_motor.brake()
right_motor.brake()
```

Tant que :

```
emergency_stop == True
```

les joysticks ne commandent plus les moteurs.

25. Comment réarmer ?

Nous pouvons décider que X réactive le robot.

Par exemple :

```
BUTTON_X = 305
```

Puis :

```
if event.code == BUTTON_X and event.value == 1:

    emergency_stop = False
```

Cela crée une vraie logique :

```
CARRE
=
ARRET D'URGENCE

X
=
REARMEMENT
```

26. Pourquoi est-ce intéressant ?

Le robot ne peut pas repartir simplement à cause d'un joystick qui n'est pas parfaitement centré.

L'utilisateur doit volontairement réarmer le système.

27. Variables principales

Nous aurons donc :

```
throttle = 0
steering = 0

speed_mode = MODE_NORMAL

emergency_stop = False
```

28. Recherche automatique de la manette

Nous reprenons la fonction utilisée précédemment.

```
def find_gamepad():

    for path in list_devices():

        device = InputDevice(path)

        if device.name == "Wireless Controller":
            return device

        if device.name == "PLAYSTATION(R)3 Controller":
            return device

    return None
```

29. Pourquoi mettre cela dans une fonction ?

Le programme principal devient plus lisible.

Au lieu de plusieurs lignes, nous pouvons simplement écrire :

```
gamepad = find_gamepad()
```

30. Vérification

```
if gamepad == None:

    print("Manette non detectee")

else:

    print("Manette connectee")
```

31. Ajouter un signal sonore

Lorsque la manette est détectée :

```
ev3.speaker.beep()
```

Cela informe immédiatement l'utilisateur.

32. Lecture des événements

Notre boucle principale sera :

```
for event in gamepad.read_loop():
```

Puis nous séparons les événements :

```
boutons
```

et :

```
axes
```

33. Gestion des boutons

```
if event.type == 1:
```

Puis nous regardons le code.

34. Gestion des axes

```
elif event.type == 3:
```

Puis :

```
if event.code == AXIS_LEFT_Y:
```

ou :

```
elif event.code == AXIS_LEFT_X:
```

35. Programme complet

Voici une première version complète du robot.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

from evdev import InputDevice
from evdev import list_devices


AXIS_LEFT_X = 0
AXIS_LEFT_Y = 1

BUTTON_SQUARE = 304
BUTTON_X = 305
BUTTON_TRIANGLE = 307
BUTTON_L1 = 308
BUTTON_R1 = 309

MODE_SLOW = 300
```

```
MODE_NORMAL = 600
MODE_FAST = 900

DEAD_ZONE_RAW = 12


ev3 = EV3Brick()

left_motor = Motor(Port.B)
right_motor = Motor(Port.C)


def limit(value, minimum, maximum):

    if value > maximum:
        return maximum

    if value < minimum:
        return minimum

    return value


def convert_axis(value, max_speed):

    centered = 128 - value

    if abs(centered) < DEAD_ZONE_RAW:
        return 0

    speed = int(centered * max_speed / 127)

    return limit(speed, -max_speed, max_speed)


def drive_robot(throttle, steering, max_speed):

    left_speed = throttle + steering
    right_speed = throttle - steering

    left_speed = limit(left_speed, -max_speed, max_speed)
    right_speed = limit(right_speed, -max_speed, max_speed)

    if left_speed == 0 and right_speed == 0:

        left_motor.stop()
        right_motor.stop()

    else:

        left_motor.run(left_speed)
        right_motor.run(right_speed)


def stop_robot():

    left_motor.brake()
    right_motor.brake()


def find_gamepad():

    for path in list_devices():

        device = InputDevice(path)

        if device.name == "Wireless Controller":
```

```
            return device

        if device.name == "PLAYSTATION(R)3 Controller":
            return device

    return None


print("Recherche manette")

gamepad = find_gamepad()


if gamepad == None:

    print("Manette non detectee")

    stop_robot()

else:

    print("Manette connectee")
    print(gamepad.name)

    ev3.speaker.beep()

    throttle = 0
    steering = 0

    speed_mode = MODE_NORMAL

    emergency_stop = False


    for event in gamepad.read_loop():


        if event.type == 1:


            if event.code == BUTTON_SQUARE and event.value == 1:

                emergency_stop = True

                throttle = 0
                steering = 0

                stop_robot()

                print("ARRET URGENCE")


            elif event.code == BUTTON_X and event.value == 1:

                emergency_stop = False

                print("SYSTEME REARME")


            elif event.code == BUTTON_TRIANGLE and event.value == 1:

                ev3.speaker.beep()


            elif event.code == BUTTON_L1:

                if event.value == 1:
```

```
                speed_mode = MODE_SLOW

                print("MODE LENT")

            elif event.value == 0:

                speed_mode = MODE_NORMAL

                print("MODE NORMAL")


        elif event.code == BUTTON_R1:

            if event.value == 1:

                speed_mode = MODE_FAST

                print("MODE RAPIDE")

            elif event.value == 0:

                speed_mode = MODE_NORMAL

                print("MODE NORMAL")


    elif event.type == 3:


        if not emergency_stop:


            if event.code == AXIS_LEFT_Y:

                throttle = convert_axis(
                    event.value,
                    speed_mode
                )


            elif event.code == AXIS_LEFT_X:

                steering = convert_axis(
                    event.value,
                    speed_mode
                )

                steering = int(steering * 0.6)


            drive_robot(
                throttle,
                steering,
                speed_mode
            )
```

36. Analyse du programme

Notre programme peut maintenant faire :

```
joystick Y
=
accélération
joystick X
=
direction
L1
=
mode lent
R1
=
mode rapide
Triangle
=
klaxon
Carré
=
arrêt d'urgence
X
=
Réarmement
```

37. Premier test

Ne pose pas immédiatement le robot au sol.

Soulève le robot pour que les roues soient libres.

Teste d’abord :

```
joystick avant
joystick arrière
joystick gauche
joystick droite
```

Vérifie le sens de rotation des moteurs.

38. Si les moteurs sont inversés

Selon le montage, le robot peut :

```
reculer lorsque nous voulons avancer
```

Dans ce cas, plusieurs solutions sont possibles.

On peut inverser :

```
throttle
```

ou adapter le sens d’un moteur.

L’objectif est simplement d’obtenir un comportement cohérent avec le montage réel.

39. Deuxième test

Teste maintenant :

```
MODE LENT
```

Le robot doit réagir moins fortement.

Puis :

```
MODE RAPIDE
```

Il doit devenir plus rapide.

40. Troisième test

Teste :

```
CARRE
```

Les moteurs doivent immédiatement s’arrêter.

Ensuite, bouge le joystick.

Le robot ne doit pas repartir.

41. Réarmement

Appuie sur :

```
X
```

Puis déplace le joystick.

Le robot peut à nouveau bouger.

Nous avons donc construit un système de sécurité beaucoup plus réaliste.

42. Amélioration : obligation de centrer le joystick

Notre réarmement possède encore un défaut.

Supposons :

```
joystick complètement en avant
```

Puis :

```
X
```

Le robot pourrait repartir directement.

Une solution plus sûre consiste à exiger que les joysticks soient centrés avant de réarmer.

43. Ajouter une variable

Nous pouvons vérifier :

```
if throttle == 0 and steering == 0:
```

avant d'autoriser le réarmement.

```
elif event.code == BUTTON_X and event.value == 1:

    if throttle == 0 and steering == 0:

        emergency_stop = False

        print("SYSTEME REARME")

    else:

        print("CENTRER JOYSTICK")
```

44. Pourquoi cette sécurité ?

Dans une vraie machine, un redémarrage inattendu peut être dangereux.

Nous appliquons donc un principe simple :

```
STOP
      |
      v
centrer commandes
      |
      v
réarmer
      |
      v
nouveau mouvement
```

45. Amélioration : LED d'état

On peut utiliser les LEDs de la brique pour montrer l'état.

Par exemple :

```
vert
=
système actif
rouge
=
arrêt d'urgence
```

Selon les fonctions Pybricks disponibles dans l'environnement, on peut utiliser les fonctions lumineuses de l'EV3.

46. Amélioration : écran

Nous pouvons aussi afficher :

```
NORMAL
LENT
RAPIDE
```

ou :

```
STOP URGENCE
```

sur l'écran.

Cela améliore fortement l'expérience utilisateur.

47. Mini défi 1

Afficher le mode de vitesse actuel sur l'écran EV3.

L'utilisateur doit pouvoir voir immédiatement :

```
MODE LENT
```

ou :

```
MODE NORMAL
```

ou :

```
MODE RAPIDE
```

48. Mini défi 2

Créer deux klaxons.

Par exemple :

```
Triangle
=
klaxon normal
```

et :

```
Cercle
=
klaxon grave
```

49. Mini défi 3

Ajouter un bouton permettant de faire pivoter automatiquement le robot de 180 degrés.

Cela t'oblige à combiner :

```
manette
+
moteurs
+
Temporisation
```

50. Mini défi 4

Créer une vitesse maximale configurable.

Par exemple :

```
MAX_USER_SPEED = 700
```

Même en mode rapide, le robot ne peut pas dépasser cette valeur.

51. Ajouter un capteur ultrasonique

Pour aller plus loin, nous pouvons combiner télécommande et sécurité automatique.

```
MANETTE

   |
   v

EV3

   |
   + moteur
   |
   + capteur ultrasonique
```

52. Principe

Même si l'utilisateur pousse le joystick vers l'avant, le robot refuse d'avancer si un obstacle est trop proche.

Par exemple :

```
distance < 15 cm
=
interdiction d'avancer
```

53. Architecture

```
JOYSTICK
   |
   v
commande avance
   |
   v
CAPTEUR
   |
   v
obstacle ?
 /       \
oui      non
 |        |
STOP     AVANCE
```

54. Exemple conceptuel

```
if distance < 150 and throttle > 0:

    throttle = 0
```

La valeur exacte et l'unité doivent être adaptées à l'API utilisée.

55. Pourquoi ce bonus est intéressant ?

Le robot devient :

```
piloté par humain
+
protégé automatiquement
```

Cette combinaison est très fréquente dans les systèmes réels.

56. Défi final : parcours complet

Construire un parcours comportant :

```
départ
ligne droite
slalom
passage étroit
marche arrière
stationnement
arrivée
```

Le pilote doit utiliser intelligemment les trois modes.

57. Mode rapide

Utilisation recommandée :

```
ligne droite
```

58. Mode normal

Utilisation recommandée :

```
virages
slalom
```

59. Mode lent

Utilisation recommandée :

```
passage étroit
stationnement
```

60. Épreuve de sécurité

Pendant le parcours, si l'enseignant annonce :

```
STOP
```

Tu dois immédiatement utiliser :

```
Carré
```

Le robot doit s'immobiliser.

Ensuite :

```
centrer joystick
réarmer
reprendre
```

61. Situation problème

Le robot bouge légèrement sans toucher au joystick.

Quelle partie faut-il vérifier ?

Correction

La zone morte.

Il faut éventuellement augmenter :

```
DEAD_ZONE_RAW
```

62. Situation problème

Le robot tourne beaucoup trop vite.

Que peut-on modifier ?

Correction

Réduire le coefficient :

```
steering = int(steering * 0.6)
```

Par exemple :

```
steering = int(steering * 0.4)
```

63. Situation problème

L'utilisateur appuie sur Carré mais le robot repart immédiatement.

Quelle est probablement la cause ?

Correction

Le programme continue à traiter les axes sans verrouillage.

Il faut utiliser :

```
emergency_stop
```

64. Situation problème

L1 est relâché alors que R1 est encore appuyé.

Le programme repasse en mode normal.

Pourquoi cela peut il poser problème ?

Correction

Notre version simple ne mémorise pas séparément l'état de L1 et R1.

Pour une gestion très précise, il faudrait mémoriser :

```
slow_pressed
fast_pressed
```

puis calculer le mode à partir de ces états.

65. Version plus robuste des modes

```
slow_pressed = False
fast_pressed = False
```

Lors de L1 :

```
slow_pressed = event.value == 1
```

Lors de R1 :

```
fast_pressed = event.value == 1
```

Puis :

```
if fast_pressed:

    speed_mode = MODE_FAST

elif slow_pressed:

    speed_mode = MODE_SLOW

else:

    speed_mode = MODE_NORMAL
```

66. Pourquoi cette version est meilleure ?

Le programme connaît réellement l'état des deux boutons.

Il ne dépend plus uniquement du dernier événement reçu.

67. Notion importante : état et événement

Nous venons de rencontrer une distinction importante.

Un :

```
événement
```

dit :

```
quelque chose vient de se produire
```

Un :

```
état
```

dit :

```
quelle est la situation actuelle
```

Exemple :

```
événement
=
L1 vient d'être pressé
état
=
L1 est actuellement maintenu
```

68. Pourquoi cette notion est importante ?

Les vrais systèmes interactifs utilisent énormément cette logique.

Par exemple :

```
clavier
manette
interface graphique
robot
jeu vidéo
```

69. Défi supplémentaire : compteur de distance

Pour les étudiants avancés, utiliser les angles des moteurs pour estimer la distance parcourue.

Afficher par exemple :

```
Distance :
3.4 m
```

Cela combine :

```
géométrie
+
encodeurs moteurs
+
Programmation
```

70. Défi supplémentaire : batterie

Afficher le niveau de batterie de la brique si l'API utilisée le permet.

On pourrait ensuite déclencher :

```
BATTERIE FAIBLE
```

lorsque la tension descend sous un seuil.

71. Questions de compréhension

Question 1

Pourquoi utiliser des fonctions ?

Question 2

Pourquoi utiliser une zone morte ?

Question 3

À quoi sert `limit()` ?

Question 4

À quoi sert `emergency_stop` ?

Question 5

Pourquoi demander de centrer le joystick avant de réarmer ?

Question 6

Quelle est la différence entre événement et état ?

Correction 1

Pour rendre le programme plus lisible, organisé et réutilisable.

Correction 2

Pour ignorer les petites variations involontaires du joystick.

Correction 3

À empêcher une valeur de dépasser les limites définies.

Correction 4

À verrouiller le robot après un arrêt d'urgence.

Correction 5

Pour empêcher un redémarrage involontaire.

Correction 6

Un événement correspond à un changement qui vient de se produire, alors qu'un état représente la situation actuelle.

72. Petit exercice écrit

Complète :

```
Le joystick Y contrôle principalement _____.

Le joystick X contrôle principalement _____.

Le bouton Carré sert d'_____.

L1 active le mode _____.

R1 active le mode _____.

Une zone morte empêche le robot de _____ lorsque le joystick est presque centré.
```

Correction

```
Le joystick Y contrôle principalement l'avance et le recul.

Le joystick X contrôle principalement la direction.

Le bouton Carré sert d'arrêt d'urgence.

L1 active le mode lent.

R1 active le mode rapide.

Une zone morte empêche le robot de bouger lorsque le joystick est presque centré.
```

73. Méthode de dépannage

Lorsque le robot ne fonctionne pas :

1. Bluetooth fonctionne t-il ?
2. La manette est-elle visible dans Linux ?
3. Les événements arrivent-ils ?
4. Les bons axes sont-ils utilisés ?
5. Les valeurs sont-elles correctes ?
6. La conversion est-elle correcte ?
7. Les moteurs fonctionnent ils séparément ?
8. L'arrêt d'urgence est-il actif ?

74. Erreurs fréquentes

Erreur 1

Les axes sont incorrects.

Solution :

Refaire la table des événements.

Erreur 2

Le sens est inversé.

Solution :

Inverser le signe ou la formule de conversion.

Erreur 3

Le robot tremble au repos.

Solution :

Augmenter légèrement la zone morte.

Erreur 4

La direction est trop agressive.

Solution :

Réduire le coefficient de `steering`.

Erreur 5

Le robot repart après l’arrêt d’urgence.

Solution :

Ajouter un véritable verrouillage logiciel.

Erreur 6

Le mode rapide rend le robot difficile à contrôler.

Solution :

Réduire `MODE_FAST`.

Cours 35 : Réseau, SSH et projet final du robot connecté

Objectifs du cours

À la fin de cette leçon, tu seras capable de :

1. Comprendre ce qu'est un réseau informatique
2. Comprendre le rôle d'une adresse IP
3. Différencier Bluetooth et réseau IP
4. Comprendre le principe de SSH
5. Accéder à distance à la brique EV3
6. Comprendre les notions client et serveur dans un réseau
7. Regrouper plusieurs modes de fonctionnement dans un même programme
8. Créer un robot capable de fonctionner en mode autonome, télécommande EV3 ou manette Bluetooth
9. Organiser un programme complexe
10. Réaliser un projet final complet

Matériel nécessaire

1. Une brique LEGO Education Mindstorms EV3
2. Une carte microSD avec EV3 MicroPython
3. Un ordinateur
4. Un câble USB
5. Facultativement un adaptateur WiFi compatible
6. Une manette PS3 ou PS4 déjà associée
7. Deux moteurs
8. Au moins un capteur
9. Facultativement une deuxième brique EV3

1. Introduction

Pendant les dernières leçons, nous avons appris à faire communiquer notre EV3 avec différents appareils.

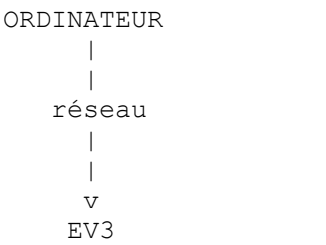
Nous avons utilisé :

```
EV3 ))) EV3
```

puis :

```
MANETTE ))) EV3
```

Nous allons maintenant découvrir une autre forme de communication :



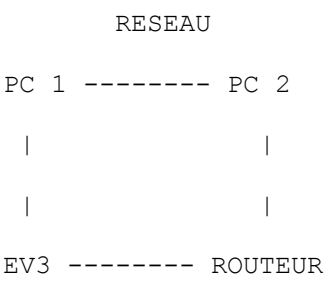
Notre brique EV3 est en réalité un petit ordinateur Linux.

Elle peut donc participer à un réseau comme un ordinateur classique.

2. Qu'est-ce qu'un réseau ?

Un réseau permet à plusieurs appareils de communiquer.

Par exemple :



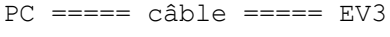
Un réseau peut contenir :

ordinateurs
serveurs
smartphones
imprimantes
robots
objets connectés

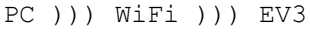
3. Réseau filaire et réseau sans fil

Comme pour les connexions précédentes, un réseau peut être filaire ou sans fil.

Réseau filaire



Réseau sans fil



Dans notre cas, l'EV3 peut notamment être relié à l'ordinateur via une connexion réseau USB.

4. Bluetooth et réseau ne signifient pas la même chose

Lors des leçons précédentes, nous utilisions Bluetooth pour connecter directement des périphériques.

Par exemple :

```
MANETTE
   |
Bluetooth
   |
   v
EV3
```

Mais un réseau IP fonctionne différemment.

```
PC
 |
réseau IP
 |
EV3
```

Bluetooth peut parfois également transporter une connexion réseau, mais il faut distinguer :

```
technologie de transport
```

et :

```
protocole réseau
```

5. Une analogie simple

On peut imaginer :

```
Bluetooth ou WiFi
=
la route
```

et :

```
IP
=
le système d'adresses permettant de savoir où aller
```

6. L’adresse IP

Sur un réseau IP, chaque appareil possède une adresse.

Par exemple :

```
PC

192.168.1.10
```

et :

```
EV3

192.168.1.20
```

On peut représenter cela ainsi :

```
192.168.1.10
     PC
      |
      |
    réseau
      |
      |
     EV3
192.168.1.20
```

7. À quoi sert l’adresse IP ?

Elle permet d’identifier une machine sur le réseau.

On peut comparer cela à une adresse postale.

```
Nom de rue
+
numéro
```

permet de trouver une maison.

Sur un réseau :

```
adresse IP
```

permet de trouver une machine.

8. Exemple

Si l’ordinateur veut communiquer avec :

```
192.168.1.20
```

il sait vers quelle machine envoyer les données.

9. Adresse locale

Les adresses comme :

```
192.168.x.x
```

sont souvent utilisées dans les réseaux locaux.

Cela signifie qu'elles servent à communiquer à l'intérieur d'un réseau privé.

Pour cette leçon, il n'est pas nécessaire d'entrer dans les détails des masques réseau ou du routage.

10. Observer l'adresse de l'EV3

Lorsque l'EV3 est relié à un ordinateur par USB et que la connexion réseau est disponible, il reçoit une adresse IP.

Selon la configuration, une adresse couramment utilisée par ev3dev via USB peut être différente d'un réseau domestique classique.

Ton objectif est surtout d'identifier :

```
EV3
=
machine avec une adresse réseau
```

11. Qu'est-ce que SSH ?

SSH signifie :

```
Secure Shell
```

SSH permet d'ouvrir un terminal sur un autre ordinateur à travers le réseau.

```
ORDINATEUR

    |
    | SSH
    |
    v

   EV3
```

12. Pourquoi SSH est utile ?

Grâce à SSH, on peut :

```
ouvrir un terminal
exécuter des commandes
observer les fichiers
lancer des programmes
diagnostiquer le système
```

sans utiliser directement l'écran et les boutons de l'EV3.

13. Local et distant

Si j'utilise directement la brique :

```
LOCAL
```

Si j'utilise mon ordinateur pour exécuter une commande sur l'EV3 à travers le réseau :

```
DISTANT
```

14. Exemple conceptuel

Depuis le PC :

```
ssh utilisateur@adresse_ip
```

Le PC ouvre alors une session distante sur la brique.

Nous ne cherchons pas ici à mémoriser une commande précise dépendant de toute configuration.

L'objectif est de comprendre le principe.

15. Architecture SSH

```
ORDINATEUR
     |
     |
 client SSH
     |
     v
==================
     réseau
==================
     |
     v
 serveur SSH
     |
     |
    EV3
```

Nous retrouvons une notion déjà connue :

```
CLIENT
```

et :

```
SERVEUR
```

16. Qui est client ?

L'ordinateur qui demande la connexion SSH joue le rôle de :

```
CLIENT
```

17. Qui est serveur ?

L'EV3 qui accepte la connexion SSH joue le rôle de :

```
SERVEUR
```

Nous retrouvons donc la même logique que lors de nos communications Bluetooth.

18. Manipulation réseau

Relie la brique EV3 à l'ordinateur.

Vérifie que la connexion réseau de la brique est active.

Repère ensuite l'adresse IP indiquée par le système.

Note :

```
Nom de la brique :

Adresse IP :

Type de connexion :
```

19. Test de communication

Une méthode classique pour tester si une machine répond sur le réseau consiste à utiliser :

```
ping
```

Le principe :

```
PC
 |
 | "Es tu là ?"
 v
EV3

 |
 | "Oui"
 v
PC
```

20. À quoi sert ping ?

`ping` permet notamment de vérifier qu'une machine est joignable.

Il ne vérifie pas que tous les services fonctionnent.

Mais il permet déjà de répondre à :

```
Est-ce que les deux machines peuvent communiquer sur le réseau ?
```

21. Activité 1

Complète :

```
PC
adresse : __________

EV3
adresse : __________
```

Puis dessine leur connexion.

22. Activité 2

Explique la différence entre :

```
Bluetooth
```

et :

```
adresse IP
```

Correction attendue

Bluetooth est une technologie de communication sans fil.

Une adresse IP permet d'identifier une machine dans un réseau utilisant IP.

23. Activité 3

Explique SSH avec tes propres mots.

Réponse possible

SSH permet d'utiliser un terminal sur une machine distante à travers le réseau.

24. Pourquoi cette notion est utile pour la robotique ?

Imagine un robot situé à plusieurs mètres.

Avec SSH :

```
ordinateur
    |
    v
réseau
    |
    v
robot
```

Nous pouvons effectuer certaines opérations sans toucher physiquement au robot.

25. Administration distante

Dans les systèmes réels, de nombreux serveurs et équipements sont administrés à distance.

Notre EV3 nous permet donc d'introduire une vraie notion d'administration système.

26. Projet final du module

Nous allons maintenant construire un robot possédant plusieurs modes de fonctionnement.

Le projet s'appellera :

ROBOT CONNECTÉ MULTIMODE

27. Les trois modes

Notre robot doit posséder :

```
MODE 1
AUTONOME
MODE 2
TELECOMMANDE EV3
MODE 3
MANETTE BLUETOOTH
```

28. Mode autonome

Dans ce mode, le robot prend ses décisions grâce aux capteurs.

```
CAPTEUR
   |
   v
EV3
   |
   v
DECISION
   |
   v
MOTEURS
```

Exemple :

```
pas d'obstacle
=
avance
obstacle
=
Tourne
```

29. Mode télécommande EV3

Dans ce mode :

```
EV3 TELECOMMANDE
        |
        |
    Bluetooth
        |
        v
     ROBOT
```

On réutilise les notions de la leçon 29.

30. Mode manette

Dans ce mode :

```
MANETTE
   |
Bluetooth
   |
   v
EV3
   |
   v
MOTEURS
```

On réutilise les leçons 31 à 34.

31. Menu au démarrage

Nous voulons afficher quelque chose comme :

```
CHOISIR MODE

1 AUTONOME

2 EV3 REMOTE

3 GAMEPAD
```

L'utilisateur choisit un mode avec les boutons de l'EV3.

32. Organisation du programme

Nous pouvons créer trois fonctions principales.

```
def autonomous_mode():
    pass
def ev3_remote_mode():
    pass
def gamepad_mode():
    pass
```

33. Pourquoi des fonctions ?

Sans fonctions, notre programme pourrait contenir des centaines de lignes difficiles à comprendre.

Avec des fonctions :

```
PROGRAMME PRINCIPAL
      |
      + autonomie
      |
      + télécommande EV3
      |
      + manette
```

La structure devient beaucoup plus claire.

34. Programme principal

Conceptuellement :

```
mode = choose_mode()

if mode == 1:
    autonomous_mode()

elif mode == 2:
    ev3_remote_mode()

elif mode == 3:
    gamepad_mode()
```

35. Fonction choose_mode()

Nous pouvons utiliser les boutons de l'EV3.

Exemple simple :

```
from pybricks.hubs import EV3Brick
from pybricks.parameters import Button
from pybricks.tools import wait

ev3 = EV3Brick()


def choose_mode():

    print("HAUT = AUTO")
    print("GAUCHE = EV3")
    print("DROITE = GAMEPAD")

    while True:

        buttons = ev3.buttons.pressed()

        if Button.UP in buttons:
            return 1

        elif Button.LEFT in buttons:
            return 2
```

```
        elif Button.RIGHT in buttons:
            return 3

        wait(50)
```

36. Résultat

```
HAUT
=
AUTONOME
GAUCHE
=
TELECOMMANDE EV3
DROITE
=
MANETTE
```

37. Mode autonome simple

Supposons que le robot possède un capteur ultrasonique.

Le principe :

```
si obstacle loin
=
avance
si obstacle proche
=
tourne
```

Pseudo code :

```
def autonomous_mode():

    while True:

        distance = sensor.distance()

        if distance > 200:
            avancer()

        else:
            tourner()
```

38. Mode télécommande EV3

Nous réutilisons la réception Bluetooth.

```
def ev3_remote_mode():

    server.wait_for_connection()

    while True:

        mailbox.wait()

        command = mailbox.read()

        if command == "AVANCE":
            avancer()

        elif command == "RECULE":
            reculer()

        elif command == "STOP":
            arreter()
```

39. Mode manette

Nous réutilisons notre programme de la leçon 34.

```
def gamepad_mode():

    gamepad = find_gamepad()

    if gamepad == None:
        return

    for event in gamepad.read_loop():

        # lecture boutons
        # lecture joysticks
        # commande moteurs
        Pass
```

40. Retourner au menu

Un vrai système multimode devrait permettre de quitter un mode.

Par exemple, le bouton central de l'EV3 peut devenir :

```
RETOUR MENU
```

Nous pouvons prévoir une variable :

```
running = True
```

Puis :

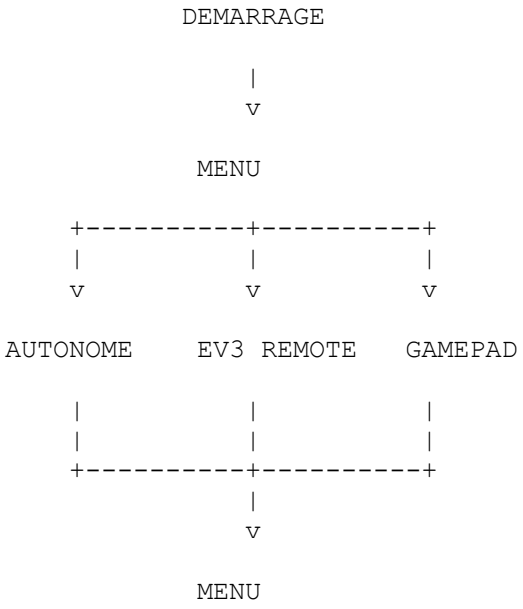
```
while running:
```

et modifier :

```
running = False
```

lorsque l'utilisateur demande de quitter.

41. Architecture finale



42. Sécurité globale

Quel que soit le mode, le robot doit toujours pouvoir être arrêté.

Nous allons donc imposer :

```
STOP
```

dans chaque mode.

Par exemple :

```
AUTONOME
=
bouton centre EV3
EV3 REMOTE
=
commande STOP
GAMEPAD
=
Carré
```

43. Pourquoi mettre la sécurité partout ?

Parce qu'un robot ne doit pas dépendre d'un seul mécanisme d'arrêt.

Chaque mode doit connaître son propre comportement de sécurité.

44. Défi 1

Créer uniquement le menu.

Le programme doit afficher le mode choisi.

Par exemple :

```
GAMEPAD SELECTED
```

sans encore lancer les moteurs.

45. Défi 2

Ajouter le mode autonome.

Le robot doit :

```
avancer
```

puis :

```
éviter un obstacle
```

46. Défi 3

Ajouter le mode télécommande EV3.

Le deuxième EV3 doit pouvoir commander :

```
AVANCE
RECULE
GAUCHE
DROITE
STOP
```

47. Défi 4

Ajouter le mode manette.

La manette doit permettre :

```
vitesse analogique
direction analogique
stop urgence
klaxon
mode lent
mode rapide
```

48. Défi 5

Permettre de revenir au menu.

Le robot ne doit pas nécessiter un redémarrage complet à chaque changement de mode.

49. Bonus : afficher le mode

L'écran peut afficher :

```
AUTO
```

ou :

```
EV3
```

ou :

```
GAMEPAD
```

pendant l'utilisation.

50. Bonus : LED selon le mode

Par exemple :

```
vert
=
autonome
orange
=
télécommande EV3
rouge
=
gamepad
```

si la version Pybricks utilisée permet ce réglage.

51. Bonus : sécurité par capteur en mode manette

Même en contrôle manuel :

```
si obstacle < 10 cm
et commande = avancer
```

alors :

```
STOP
```

Le robot devient semi autonome.

52. Bonus : retour d’état

Dans le mode télécommande EV3, le robot peut répondre :

```
BATTERY OK
```

ou :

```
OBSTACLE
```

ou :

```
MODE ACTIVE
```

vers la télécommande.

53. Bonus : journal d’événements

Le robot peut afficher :

```
MODE GAMEPAD

Vitesse 450

Direction 120

STOP

MODE AUTO
```

Ce principe ressemble à des logs.

54. Qu’est-ce qu’un log ?

Un log est une trace d’événements.

Par exemple :

```
10:02 robot démarré
10:03 gamepad connecté
10:04 mode rapide
10:05 arrêt urgence
```

Les systèmes informatiques réels utilisent énormément les logs pour comprendre ce qui s’est passé.

55. Lien avec SSH

Si le robot est accessible à distance, SSH permet de consulter et diagnostiquer ce type d’informations sans toucher directement à la brique.

Nous relions donc :

```
réseau
+
Linux
+
robotique
+
Diagnostic
```

56. Situation problème

L’ordinateur connaît l’adresse IP de l’EV3 mais SSH ne fonctionne pas.

Peut-on conclure que le réseau ne fonctionne pas ?

Correction

Pas nécessairement.

La connectivité réseau peut fonctionner alors que le service SSH rencontre un autre problème.

Il faut diagnostiquer couche par couche.

57. Situation problème

Le robot répond au réseau mais la manette ne fonctionne pas.

Quel système faut il vérifier ?

Correction

La chaîne Bluetooth et `evdev`, pas nécessairement le réseau IP.

58. Situation problème

Le mode autonome fonctionne mais pas le mode télécommande.

Que peut-on en déduire ?

Correction

Les moteurs et une grande partie du robot fonctionnent probablement correctement.

Le problème est plus probablement lié à la communication ou à la logique du mode télécommande.

59. Méthode de diagnostic finale

Toujours séparer les couches :

```
ALIMENTATION
     |
     v
MATERIEL
     |
     v
SYSTEME
     |
     v
CONNEXION
     |
     v
PROGRAMME
     |
     v
LOGIQUE
```

60. Projet final

Le robot doit satisfaire le cahier des charges suivant.

Fonction 1

Mode autonome fonctionnel.

Fonction 2

Mode télécommande EV3 fonctionnel.

Fonction 3

Mode manette Bluetooth fonctionnel.

Fonction 4

Menu de sélection.

Fonction 5

Arrêt de sécurité dans chaque mode.

Fonction 6

Au moins un capteur utilisé.

Fonction 7

Code organisé en fonctions.

Fonction 8

Retour possible au menu.

61. Cahier des charges détaillé

Le programme doit au minimum contenir :

```
def choose_mode():
def autonomous_mode():
def ev3_remote_mode():
def gamepad_mode():
def stop_robot():
```

62. Partie autonomie

Le robot doit prendre au moins une décision grâce à un capteur.

Par exemple :

```
distance > 20 cm
=
avance
```

et :

```
distance <= 20 cm
=
Tourne
```

63. Partie télécommande EV3

Les commandes minimales sont :

```
AVANCE
RECULE
GAUCHE
DROITE
STOP
```

64. Partie manette

Les commandes minimales sont :

```
joystick vertical
=
vitesse
joystick horizontal
=
direction
Carré
=
arrêt d'urgence
```

65. Présentation finale

Chaque étudiant doit présenter :

1. l’architecture de son robot
2. les modes disponibles
3. la manière dont les communications fonctionnent
4. le mécanisme de sécurité
5. un exemple de problème rencontré
6. la méthode utilisée pour le résoudre

66. Questions de compréhension

Question 1

Qu’est ce qu’un réseau ?

Question 2

À quoi sert une adresse IP ?

Question 3

Qu’est ce que SSH ?

Question 4

Dans une connexion SSH, qui est généralement client ?

Question 5

Pourquoi utiliser plusieurs fonctions pour les différents modes ?

Question 6

Pourquoi chaque mode doit il posséder un mécanisme de sécurité ?

Correction 1

Un ensemble d’appareils capables de communiquer entre eux.

Correction 2

À identifier une machine dans un réseau IP.

Correction 3

Un protocole permettant notamment d’ouvrir une session terminal distante de manière sécurisée.

Correction 4

La machine qui initie la connexion.

Correction 5

Pour rendre le programme plus clair et plus facile à maintenir.

Correction 6

Pour pouvoir arrêter le robot quel que soit son mode de fonctionnement.

67. Petit exercice écrit

Complète :

```
Une adresse ______ permet d'identifier une machine sur un réseau.

SSH permet d'ouvrir un ______ distant.

Le robot possède trois modes : ______, ______ et ______.

Les différentes parties du programme sont organisées en ______.

Un système complexe doit être diagnostiqué couche par ______.
```

Correction

```
Une adresse IP permet d'identifier une machine sur un réseau.

SSH permet d'ouvrir un terminal distant.

Le robot possède trois modes : autonome, télécommande EV3 et manette Bluetooth.

Les différentes parties du programme sont organisées en fonctions.

Un système complexe doit être diagnostiqué couche par couche.
```

68. Erreurs fréquentes

Erreur 1

Essayer de tout programmer d'un coup.

Solution :

Tester chaque mode séparément.

Erreur 2

Un mode bloque et empêche le retour au menu.

Solution :

Prévoir une condition de sortie dans chaque boucle.

Erreur 3

Les mêmes moteurs sont commandés par plusieurs modes en même temps.

Solution :

Un seul mode doit être actif à la fois.

Erreur 4

La sécurité n'est pas commune aux trois modes.

Solution :

Créer une fonction `stop_robot()` réutilisée partout.

Erreur 5

Le code devient impossible à lire.

Solution :

Créer des fonctions plus petites et nommées clairement.

69. Compétences acquises dans la partie 2

Au cours des leçons 26 à 35, tu auras découvert :

USB

Bluetooth

WiFi

Linux

ev3dev

client

serveur

mailbox

périphériques d'entrée

/dev/input

evdev

événements

boutons

joysticks

valeurs analogiques

zone morte

pilotage proportionnel

sécurité

réseau IP

SSH
Réseau et robot multimode

70. Conclusion du module

Le point le plus important de ce module est que l'EV3 n'est plus considéré uniquement comme :

```
une brique LEGO programmable
```

mais comme :

```
un ordinateur Linux
+
des capteurs
+
des moteurs
+
des interfaces réseau
+
des périphériques externes
```

Tu passes ainsi progressivement de la programmation LEGO à des concepts utilisés dans la vraie informatique, l'administration système, les réseaux et la robotique.

Avec cette leçon 35, le **module Connectivité est complet** et notre cours passe donc de **25 à 35 leçons**

Conclusion générale

Plus de 900 pages de cours, ce n'est pas rien !

Je te félicite pour ta ténacité si tu es arrivé au bout.

J'espère que ce cours t'a plu et t'a donné envie d'aller plus loin dans la robotique et/ou la programmation.

Tu peux toujours envoyer un feedback et des suggestions sur patrick.hainaut@gmail.com

Bien sûr, maintenant, l'intelligence artificielle existe et programme à notre place.

Ce cours a d'ailleurs été conçu, en partie, à l'aide de l'I.A.

Le but de ce cours n'est pas de te faire bêtement « pisser du code » mais de t'apprendre à réfléchir sur des problématiques, élaborer des hypothèses, trouver des solutions et les expérimenter.

Quand on sait faire ça, alors, on peut utiliser l'I.A. correctement et garder le contrôle.

L'avantage avec Lego Education, c'est qu'on monte très rapidement des robots de toute sorte et que l'on peut se concentrer sur la programmation.

On peut, pour les débutants, commencer par la programmation par blocs mais c'est avec MicroPython que toute la puissance de la brique EV3 se révèle.

De plus, Python est un langage très utilisé car assez facile à prendre en main et utilisable dans beaucoup de situations, c'est donc une opportunité de l'apprendre.

Ce qui a été vu ici, avec la brique Mindstorms EV3 peut être adapté à la brique Spike, le programme Lego Education qui a succédé à Mindstorms.

Bonne continuation dans le domaine de la robotique,

Patrick Hainaut

