Ecole Robots
Apprendre à coder avec les robots éducatifs

Kit de programmation
CURSUS EDUCATION NATIONALE

Python - Niveau lycée

Les bases de la programmation

La syntaxe Python - partie 2

Prénom : _____ Nom : _____ Classe : _____

ARTEC
Speechi

Les programmes nécessaires à la réalisation des robots sont disponibles en téléchargement sur le site www.ecolerobots.com.

Toutes les boîtes et les pièces détachées sont aussi disponibles sur le site www.ecolerobots.com.

Les bases de la programmation
La syntaxe Python - partie 2

Montage, programmation, robotique.
École Robots - Cursus Éducation Nationale

Ecole Robots
Apprendre à coder avec les robots éducatifs

Sommaire

Chapitre 1 : Les séquences et les boucles

1. Ce que vous allez apprendre dans cette leçon

En programmation, il existe deux types de structures de contrôle : les boucles et les alternatives. Cette leçon porte sur les boucles qui peuvent être utilisées pour parcourir les éléments d'une séquence. Des données autres que les nombres et chaînes de caractères y seront également décrites.

À la fin de ce chapitre, nous vous montrerons comment programmer le buzzer à jouer une mélodie en écrivant un code concis, grâce aux traitements de base.

2. Les séquences en Python

Découvrons d'abord les données appelées « séquence ». Une séquence désigne un ensemble de valeurs ordonnées qu'elle traite d'un seul tenant.

En Python, les éléments suivants sont des séquences :

- Liste
- Tuple
- Range : liste de valeurs numériques ordonnées (sens croissant ou décroissant) générée par la fonction range().
- Chaîne de caractères : une chaîne est considérée comme une séquence de caractères.

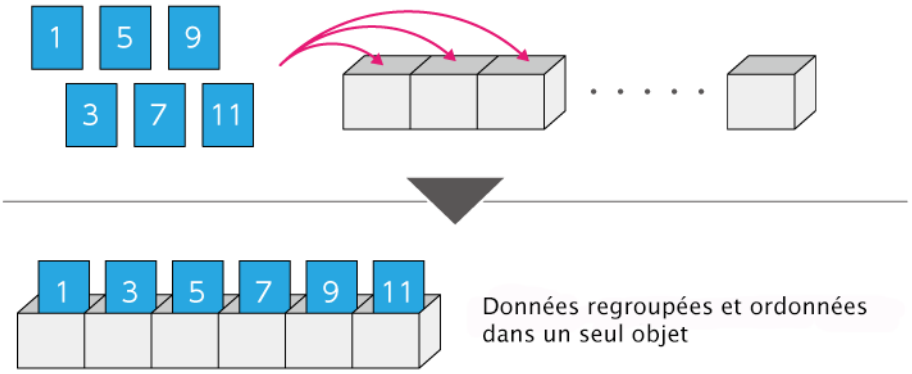
Les deux éléments suivants ne sont pas des séquences au sens strict, mais des types de données qui peuvent être manipulées comme des séquences :

- Dictionnaire
- Ensemble

Dans les parties qui suivent, nous allons examiner de plus près les listes, les tuples et les dictionnaires.

2.1. Les listes

Une liste est un type de données qui stocke plusieurs éléments les uns à la suite des autres. Ces éléments n'ayant eux-mêmes pas forcément à être tous de même type, la liste s'avère très pratique pour le cas où vous souhaiteriez gérer un ensemble de données hétéroclites (par exemple composé de nombres et de chaînes de caractères) puisqu'elle permet de stocker plusieurs valeurs dans un unique objet, de manière ordonnée.



■ Définir une liste

On crée une liste en mettant les différents éléments qui la composent entre crochets [ ] et en les séparant par des virgules :

```
nomDeLaListe = [élément 0, élément 1, élément 2, ...]
```

Exécutez, par exemple, le code suivant dans le terminal pour créer une liste qui répertorie les aliments stockés dans le réfrigérateur.

```
>>> aliments = ['lait', 'viande', 'oeufs', 'carotte']
```

Vous pouvez vérifier le contenu de la liste créée en utilisant print().

```
>>> print(aliments)
['lait', 'viande', 'oeufs', 'carotte']
```

■ Accéder à des éléments particuliers d'une liste

Pour accéder au contenu d'un élément d'une liste, il suffit d'écrire le nom de la liste immédiatement suivi du numéro de la position dudit élément entre crochets [ ].

```
nomDeLaListe[numéro]
```

Les éléments stockés dans la liste sont comptés à partir de 0 : position 0, 1, 2, etc. Spécifions, dans l'ordre, les positions de la liste donnée précédemment pour en consulter les éléments.

```
>>> print(aliments[0])
lait
>>> print(aliments[1])
viande
>>> print(aliments[2])
oeufs
>>> print(aliments[3])
carotte
```

En spécifiant le numéro de position « -1 », vous récupérerez le dernier élément de la liste.

```
>>> print(aliments[-1])
carotte
```

■ Accéder à plusieurs éléments d'une liste

Pour accéder à plusieurs éléments d'une liste, indiquez une tranche de la façon suivante :

```
nomDeLaListe[position de départ : position de fin]
```

À titre d'exemple, extrayons de la liste *aliments* la tranche d'éléments allant des positions 0 à 2.

```
>>>print(aliments[0:3])
['lait', 'viande', 'oeufs']
```

Dans le code ci-dessus, la position de départ spécifiée est 0 et la position finale, 3. Comme on peut le constater dans le résultat qui s'affiche après exécution, l'élément dont le numéro de position est spécifié à la fin n'est pas extrait. De manière générale, la syntaxe liste[n,m] renvoie une liste composée des éléments de la liste initiale allant de la position n à m-1.

Il est également possible d'extraire des éléments en spécifiant une incrémentation dans une tranche.

```
nomDeLaListe [Position de départ : Position de fin : Incrément]
```

L'incrément indique de combien de cases se déplacer dans la liste pour aller chercher l'élément suivant. Si on précise un incrément égal à n, on récupère le premier élément, puis celui à l'incrément n cases plus loin, et ainsi de suite.

```
liste[position de départ], liste[position de départ + 1 x incrément],
liste[position de départ + 2 x incrément], etc.
```

Par exemple, si l'incrément est égal à 4, les éléments suivants seront récupérés :

```
liste[position de départ], liste[position de départ + 4],
liste[position de départ + 8], etc.
```

Encore une fois, vérifions en exécutant le code suivant :

```
>>> print(aliments[0:4:2])
['lait', 'œufs']
```

■ Utilisation des méthodes de liste

Des éléments peuvent également être ajoutés et supprimés d'une liste à l'aide de méthodes.

Nom de la méthode	**Fonction**
append()	Ajouter un nouvel élément à la fin de la liste.
pop()	Supprimer l'élément dont la position est spécifiée dans la méthode.
remove()	Supprimer l'élément dont la valeur est spécifiée dans la méthode.

Ajoutons, d'abord, l'élément « bacon » à notre liste d'aliments en utilisant la méthode append().

```
>>> aliments.append('bacon')
>>> print(aliments)
['lait', 'viande', 'oeufs', 'carotte', 'bacon']
```

Supprimons à présent l'élément « œufs » de notre liste d'aliments avec la méthode pop(). Comme l'élément « œufs » se trouve en 2e position de la liste, supprimons-le avec le code suivant :

```
>>> aliments.pop(2)
'oeufs'
>>> print(aliments)
['lait', 'viande', 'carotte', 'bacon']
```

Après l'exécution de `aliments.pop(2)`, `'oeufs'` s'affiche dans le terminal parce que la méthode pop() renvoie l'élément supprimé de la liste en valeur de retour.

Les listes dont des éléments sont supprimés sont renumérotées à partir de 0. Pour cette raison, la prochaine fois que vous voudrez accéder au contenu de l'élément en 2e position, c'est l'élément « carotte » qui s'affichera dorénavant.

```
>>> print(aliments[2])
carotte
```

Pour supprimer un élément spécifique avec la méthode pop(), il faut se rappeler le numéro de cet élément, ce qui peut ne pas s'avérer pratique si la liste est longue. La méthode remove(), quant à elle, permet de supprimer un élément en spécifiant sa valeur.

```
>>> aliments.remove('carotte')
>>> print(aliments)
['lait', 'viande', 'bacon']
```

Contrairement à la méthode pop(), la méthode remove() ne renvoie aucune valeur de retour.

2.2. Les tuples

Les tuples sont des données qui ressemblent aux listes. Cependant, à la différence des listes qui peuvent être modifiées en ajoutant ou en supprimant des éléments, un tuple n'est pas modifiable - on dit que c'est un objet immuable. Une fois qu'un tuple est créé, aucun des éléments qui le constituent ne peut être ni remplacé, ni ajouté ou supprimé.

À première vue, l'utilisation des tuples semble donc limitée par rapport à celle des listes. Deux raisons expliquent néanmoins l'intérêt des tuples en Python :

1. L'usage d'un tuple plutôt que d'une liste peut permettre, dans certains cas, une économie de place mémoire et des temps d'exécution plus rapides.

En effet, d'une manière générale, un objet immuable permet d'obtenir une copie de celui-ci en donnant seulement sa référence* sans avoir besoin de le copier véritablement. Cela implique donc une rapidité d'exécution plus grande et une économie de la mémoire. Cependant, dans le cadre d'une utilisation personnelle impliquant des calculs de faibles volumes, cet avantage s'avère négligeable.

* Référence : la valeur indiquant son emplacement mémoire.

2. Le tuple peut servir de valeur constante au cours de l'exécution du programme.

Par exemple, définir le tuple piece = ("pile", "face") permet par la suite de transformer facilement l'affichage d'un 0 ou 1 par pile ou face sans utiliser de *if*.

Nous avons déjà expliqué l'importance de bien nommer une variable ou une fonction pour comprendre rapidement à quoi elles servent. Même si une valeur ne varie pas, comme la valeur Pi

ou la durée d'une journée, il est plus facile de comprendre ce que la valeur représente en la stockant dans une variable que vous aurez nommée.

Les variables dont les valeurs sont fixes dans le programme sont appelées « constantes ». De ce point de vue, le tuple est semblable à une constante, puisqu'il permet de gérer une suite de valeurs fixes.

■ Définir un tuple

Pour créer un tuple, utilisez des parenthèses ( ) en séparant les éléments constituant le tuple comme suit :

```
nomDuTuple = (élément 1, élément 2, élément 3, ...)
```

Listons, par exemple, dans un tuple, les 5 sens de l'être humain.

```
>>> sens = ('vue', 'odorat', 'toucher', 'ouïe', 'goût')
```

Comme pour les listes, les tuples peuvent être affichés par la fonction print().

```
>>> print(sens)
('vue', 'odorat', 'toucher', 'ouïe', 'goût')
```

■ Extraire les éléments d'un tuple

Comme dans les listes, les éléments d'un tuple sont indexés à partir de 0. Pour accéder à l'un de ses éléments, spécifiez le numéro de sa position entre crochets [ ].

```
>>> print(sens[2])
toucher
```

Si vous ne pouvez pas modifier les éléments d'un tuple, en revanche, vous pouvez bien évidemment remplacer la variable qui stocke le tuple.

```
>>> a = (1, 2, 3)
>>> print(a)
(1, 2, 3)
>>> a = (4, 5, 6)
>>> print(a)
(4, 5, 6)
```

Attention ! Si vous ne stockez qu'un seul élément dans un tuple, il faut adopter une syntaxe différente. Par exemple, s'il est déclaré comme suit, il ne sera pas traité comme un tuple :

```
>>> b = (1)
>>> print(b)
1
```

Les parenthèses () servant également aux opérations arithmétiques, b est simplement traité ici comme une variable contenant la valeur entière 1 - ce qui est logique si l'on pense que l'on peut affecter directement un nom de variable à un calcul, par exemple c = (1+4)*2, ou même b = (1+0)*1, ce qui est nécessaire dans le cadre de la programmation. Pour signifier que l'on veut stocker dans la variable b un tuple contenant l'unique entier 1, il suffit d'ajouter une virgule après le « 1 ».

```
>>> b = (1,)
```

```
>>> print(b)
(1,)
```

2.3. Le dictionnaire

Dans les listes et les tuples, chaque élément est automatiquement indexé dans un ordre croissant, en partant de 0, et l'on peut accéder aux éléments les composant en spécifiant leur position.

Les dictionnaires permettent, quant à eux, d'indexer des éléments non plus simplement par des chiffres, mais par des index appelés « clés », définis directement par l'utilisateur. Ces clés peuvent être des entiers, des chaînes de caractères ou des tuples ne contenant que des objets immuables.

■ Définir un dictionnaire

Pour créer un dictionnaire, utilisez des accolades {} et associez une clé à chaque valeur à l'aide de deux points « : », les différents couples « clé :valeur » étant séparés les uns des autres par des virgules, comme montré ci-dessous.

```
nomDuDictionnaire = {Clé 1: Élément 1, Clé 2: Élément 2, Clé 3: Élément 3 ...}
```

Dans un dictionnaire, une clé doit être unique. Si deux ou plusieurs éléments sont enregistrés avec la même clé, les premiers éléments attribués à la clé seront écrasés par le dernier.

À titre d'exemple, créons un dictionnaire appelé « contact » et enregistrons des valeurs pour chacune des trois clés que nous nommerons « prénom », « nom » et « no_tél ».

```
>>> contact = {'prénom' : 'Thomas', 'nom' : 'Dubois', 'no_tél' : '01-23-45-67-89'}
>>> print(contact)
{'prénom': 'Thomas', 'nom': 'Dubois', 'no_tél': '01-23-45-67-89'}
```

Les clés d'un dictionnaire ne constituant pas un ensemble ordonné, le résultat affiché par la fonction print() ne correspond pas nécessairement à l'ordre donné aux éléments au moment de la création. De manière générale, il ne faut pas se fier à l'ordre apparent des clés d'un dictionnaire.

■ Accéder aux valeurs du dictionnaire

Pour accéder à la valeur d'un élément d'un dictionnaire, spécifiez la clé de l'élément recherché entre crochets [ ]. Comme dans un dictionnaire papier, nous recherchons un nom (une clé) pour trouver sa définition (sa valeur).

```
nomDuDictionnaire [clé]
```

Accédons à chaque élément du dictionnaire « contact » créé précédemment et affichons-les.

```
>>> print(contact['prénom'])
Thomas
>>> print(contact['nom'])
Dubois
>>> print(contact['no_tél'])
01-23-45-67-89
```

Les clés d'un dictionnaire peuvent être, comme on l'a dit, des valeurs entières, des chaînes ou encore des tuples, pourvu que ces derniers ne contiennent eux-mêmes que des éléments immuables. Quand l'énumération 0, 1, 2, … est utilisée en guise de clés, alors le dictionnaire ne diffère pas de la

liste. Selon l'utilisation désirée, le dictionnaire peut donc parfois être préféré à la liste s'il s'avère plus pratique.

■ Ajouter des éléments au dictionnaire

Pour ajouter une nouvelle clé à un dictionnaire, aucune méthode n'est nécessaire. Il suffit d'écrire l'instruction suivante :

```
nomDuDictionnaire [clé] = valeur associée à la clé
```

Ajoutons, par exemple, au dictionnaire « contact » un code postal et affichons-le avec la fonction print().

```
>>> contact['code_postal']='59000'
>>> print(contact)
{'code_postal': '59000', 'nom': 'Dubois', 'prénom': 'Thomas',
'no_tél': '01-23-45-67-89'}
```

■ Utiliser les méthodes du dictionnaire

Pour extraire un élément du dictionnaire tout en le supprimant, utilisez la méthode pop() comme pour la liste. Supprimons, par exemple, le code postal que nous venons d'ajouter au dictionnaire.

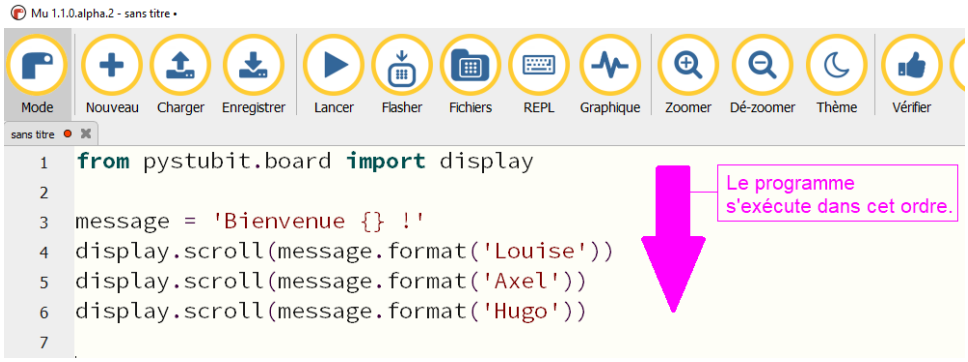
```
>>> contact.pop('code_postal')
'59000'
>>> print(contact)
{'nom': 'Dubois', 'prénom': 'Thomas', 'no_tél': '01-23-45-67-89'}
```

Comme vous pouvez le constater, la méthode du dictionnaire pop() renvoie l'élément supprimé en valeur de retour.

3. Traitement séquentiel

Le traitement séquentiel consiste à exécuter des instructions dans un ordre prédéterminé. En Python, le code s'exécute en partant de la ligne supérieure vers les lignes inférieures. Tous les programmes créés dans les leçons précédentes ont également été traités séquentiellement.

Exemple : Programme de bienvenue créé au chapitre 5 du manuel n°1



4. Traitement itératif

Un processus itératif est un processus qui répète un bloc d'instructions spécifié. Il existe deux types d'itérations :
- la boucle infinie qui répète le processus indéfiniment,
- la boucle conditionnelle qui répète le processus tant qu'une certaine condition est remplie.

Ces deux types d'itérations sont, la plupart du temps, décrites par les boucles *while* et *for*. Elles sont utiles pour raccourcir des programmes qui fonctionnent de façon séquentielle avec des données séquencées comme les listes ou les tuples.

4.1. Le traitement itératif avec la boucle *while*

L'instruction *while* fonctionne ainsi : tant que l'expression conditionnelle écrite à droite de *while* et avant les deux-points est satisfaite, le bloc d'instructions qui dépend d'elle est exécuté à plusieurs reprises.

```
while expression conditionnelle :
        Bloc d'instructions à répéter
        .
        .
        .
```

Comme pour les fonctions (*cf.* chapitre 4 du manuel n°1), le bloc d'instructions dépendant de *while* est indenté, c'est-à-dire mis en retrait.

Les expressions conditionnelles utilisent les symboles « = », « > » et « < », appelés opérateurs de comparaison.

Exemples d'opérateurs de comparaison utilisés dans les expressions conditionnelles :

Opérateur de comparaison	Exemple d'utilisation	Signification
==	a == b	a et b sont identiques
!=	a != b	a et b sont différents
<	a < b	a est strictement inférieur à b (exclut le cas a == b)
>	a > b	a est strictement supérieur à b (exclut le cas a == b)
<=	a <= b	a est inférieur ou égal à b (inclut le cas a == b)
>=	a >= b	a est égal ou supérieur à b (inclut le cas a == b)

Notez que l'opérateur d'égalité utilise deux symboles égal « == » au lieu d'un seul, car un seul « = » signifie l'affectation.

■ Traitement itératif utilisant des expressions conditionnelles

À titre d'exemple, créons un processus qui compte de 0 à 9 en combinant une instruction *while* et une expression conditionnelle à l'aide d'un opérateur de comparaison. Recopiez le code suivant dans la zone d'édition. Veillez à ne pas oublier les deux points « : » après l'expression conditionnelle.

[Exemple de code 4-1-1]

```
1  compte = 0
2  while compte < 10:
3      print(compte)
4      compte +=1
```

- À la ligne 2, se trouve l'expression conditionnelle `while compte < 10`. Le bloc d'instructions indenté est répété tant que la valeur de la variable est inférieure à 10.
- À la ligne 4, la ligne d'instruction `compte +=1`, qui équivaut à « compte = compte + 1 », incrémente la valeur de la variable de 1 chaque fois qu'elle est exécutée. Inversement, pour diminuer de 1 la valeur de la variable, vous pouvez écrire `compte -=1` au lieu de « compte = compte – 1 ».

Exécutez ce programme et vérifiez que les chiffres s'affichent dans l'ordre, de 0 à 9, dans le terminal, comme suit :

```
0
1
2
3
4
5
6
7
8
9
```

4.2. Le traitement itératif avec la boucle *for*

L'instruction *for* peut être utilisée en combinaison avec des listes, des tuples ou encore la fonction range(). Par exemple, si vous remplacez, dans le programme précédent, l'instruction *while* par une instruction *for*, voici à quoi cela ressemble :

[Exemple de code 4-2-1]

```
1  for compte in range(10):
2      print(compte)
```

Exécutez ce programme et vérifiez que le résultat obtenu est identique à celui obtenu avec l'instruction *while*.

Ce programme peut sembler difficile en raison de l'opérateur *in* et de la fonction range(). Pour comprendre son fonctionnement, expliquons le rôle de chacun.

■ Rôle de la fonction range()

La fonction range() génère une séquence de valeurs dont les caractéristiques sont spécifiées en argument. La fonction range() comportent trois arguments :

```
range(valeur entière de départ, valeur entière finale, incrément)
```

Cette fonction permet de générer une séquence d'entiers en partant d'une « valeur entière de départ » qui s'incrémente pour arriver à une « valeur entière finale ».

Les deux arguments « valeur entière de départ » et « incrément » sont optionnels. Par défaut, la valeur entière de départ est de « 0 » et l'incrément de « 1 ». La « valeur entière finale », quant à elle, n'est pas incluse dans la séquence des entiers générés. Par conséquent, si l'incrément est égal à 1, la séquence d'entiers démarrera de la valeur entière de départ et finira à la valeur entière finale moins un.

La séquence de valeurs générée par la fonction range() est traitée comme un objet spécial appelé « itérable ». Si vous exécutez le code suivant dans le terminal, vous constaterez que la valeur de retour n'est ni une liste ni un tuple.

```
>>> range(10)
range(0, 10)
```

■ Rôle de l'opérateur *in*

Dans l'instruction *for*, l'opérateur *in* sert à parcourir un à un les éléments d'une liste, d'un tuple, d'un dictionnaire ou d'une tranche. Par exemple, l'instruction `for compte in range(10):` fera parcourir à la variable « compte » une séquence de nombres entiers allant de 0 à 9.

Note : L'opérateur *in* peut également être utilisé pour vérifier si un élément d'une valeur spécifique existe dans une liste, un tuple ou un dictionnaire.

À la lumière des explications données, observons à nouveau l'exemple de [code 4-2-1].

```
1  for compte in range(10):
2      print(compte)
```

Dans ce code, l'opérateur *in* parcourt, un à un et dans l'ordre, la séquence d'entiers générée par la fonction range(10). Ces entiers sont stockés dans la variable « compte », tandis que le bloc d'instructions indenté s'exécute de manière répétée pour afficher chaque entier dans le terminal jusqu'à ce que soit terminée l'extraction de chaque entier.

Maintenant que vous avez compris le comportement de la fonction range(), vous pouvez facilement créer un programme qui compte à rebours de 9 à 0.

[Exemple de code 4-2-2]

```
1  for compte in range(9, -1, -1):
2      print(compte)
```

(Résultat après exécution)

```
9
8
7
6
5
4
3
2
1
0
```

En spécifiant un entier négatif en guise d'incrément, vous créez une séquence d'entiers qui va dans l'ordre inverse, ici « 9, 8, 7, 6, ... ».

4.3. Le traitement itératif à l'aide de listes et de tuples

Comme mentionné dans la partie précédente, l'opérateur *in* peut servir à parcourir les éléments d'une liste, d'un tuple ou d'un dictionnaire dans l'ordre.

■ Accéder aux éléments d'une liste dans l'ordre

Dans l'exemple suivant, les éléments de la liste « fruits » sont parcourus séquentiellement et stockés tour à tour dans la variable « fruit » avant de s'afficher dans le terminal grâce à la fonction print().

[Exemple de code 4-3-1]

```
1 fruits = ['pomme', 'banane', 'cerise', 'orange']
2 for fruit in fruits:
3     print(fruit)
```

(Résultat après exécution)

```
pomme
banane
cerise
orange
```

■ Accéder aux éléments d'un tuple dans l'ordre

Les éléments d'un tuple sont parcourus de la même manière que ceux d'une liste. Si l'on remplace donc la liste « fruits » par un tuple « fruits », nous obtiendrons le même résultat.

[Exemple de code 4-3-2]

```
1 fruits = ('pomme', 'banane', 'cerise', 'orange')
2 for fruit in fruits:
3     print(fruit)
```

(Résultat après exécution)

```
pomme
banane
cerise
```

```
orange
```

■ Accéder aux éléments d'un dictionnaire

Dans le cas d'un dictionnaire, en revanche, c'est la clé qui sera renvoyée si vous utilisez la même instruction que celle de la liste et du tuple. Vous noterez que, dans le dictionnaire, l'ordre lors de l'initialisation ne correspond pas à l'ordre de parcours.

[Exemple de code 4-3-3]

```
fruits = {'pomme':120, 'banane':80, 'cerise':300, 'orange':100}
for key in fruits:
    print(key)
```

(Résultat après exécution)

```
cerise
orange
banane
pomme
```

Si vous souhaitez récupérer les valeurs plutôt que les clés, alors il faut utiliser la méthode values(), comme suit :

[Exemple de code 4-3-4]

```
fruits = {'pomme':120, 'banane':80, 'cerise':300, 'orange':100}
for value in fruits.values():
    print(value)
```

(Résultat après exécution)

```
300
100
80
120
```

Pour récupérer la clé et la valeur, utilisez la méthode items(). Elles seront récupérées sous forme de tuple.

[Exemple de code 4-3-5]

```
fruits = {'pomme':120, 'banane':80, 'cerise':300, 'orange':100}
for item in fruits.items():
    print(item)
```

(Résultat après exécution)

```
('cerise', 300)
('orange', 100)
('banane', 80)
('pomme', 120)
```

5. Défi : jouer une mélodie définie à l'aide de tuples

Réutilisons les tuples et les instructions apprises dans cette leçon pour écrire un programme simple qui joue des morceaux sur le buzzer de l'ESPeRobo.

5.1. Création d'un programme

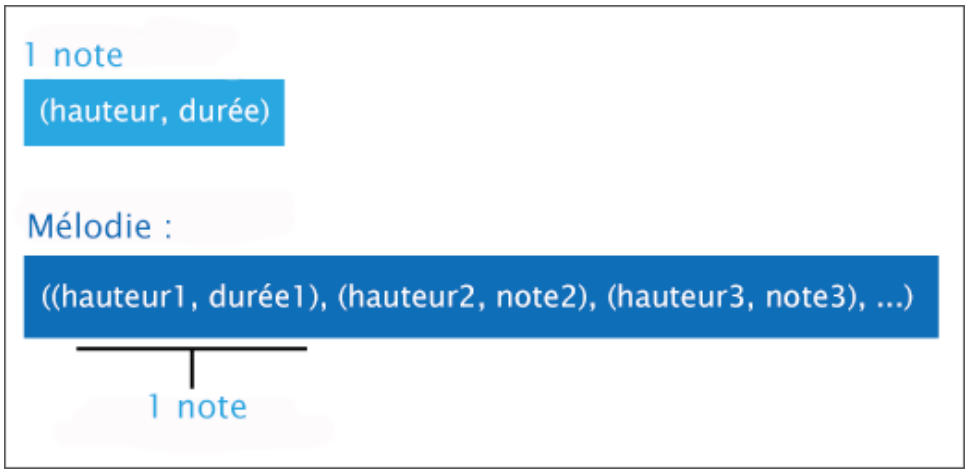
Pour pouvoir utiliser le buzzer de l'ESPeRobo, recopiez le code ci-dessous dans la zone d'édition.

```
1 from pystubit.board import buzzer
2 import time
```

Nous allons, à présent, organiser les données de notre morceau en tuples. Comme :

- une mélodie est une suite de notes,
- une note a une hauteur et une durée,

nous utiliserons une structure double qui prendra la forme d'un tuple imbriqué dans un autre tuple pour mettre au point notre mélodie.



Commençons par une mélodie simple qui joue les hauteurs de note "C4" à "C5" dans l'ordre.

Numéro	Hauteur de note	Durée
0	C4 (Do)	600
1	D4 (Ré)	600
2	E4 (Mi)	600
3	F4 (Fa)	600
4	G4 (Sol)	600
5	A4 (La)	600
6	B4 (Si)	600
7	C5 (Do)	600

Le code suivant reproduit, sous la forme d'un tuple, la mélodie présentée dans le tableau ci-dessus. Quand vous créez des tuples, des listes ou des dictionnaires, nous vous conseillons d'écrire un élément par ligne pour une meilleure lisibilité.

```
from pystubit.board import buzzer
import time

melodie = (
        ('C4', 600),
        ('D4', 500),
        ('E4', 400),
        ('F4', 600),
        ('G4', 500),
        ('A4', 400),
        ('B4', 600),
        ('C5', 500),
    )
```

À l'aide d'une instruction *for*, extrayez dans l'ordre les éléments des tuples définis. Dans le code suivant, les deux éléments composant le tuple sont stockés dans la variable « note », la hauteur de la note est extraite par « note [0] » et la longueur de la note par « note [1] ».

[Exemple de code 5-1-1]

```
from pystubit.board import buzzer
import time

melodie = (
        ('C4', 600),
        ('D4', 500),
        ('E4', 400),
        ('F4', 600),
        ('G4', 500),
        ('A4', 400),
        ('B4', 600),
        ('C5', 500),
    )

for note in melodie:
    buzzer.on(note[0])
    time.sleep_ms(note[1])
buzzer.off()
```

Notez que la ligne 18 n'est pas indentée parce qu'on souhaite arrêter le buzzer une fois toutes les notes jouées et non à chaque fois qu'une note est jouée afin que la mélodie soit fluide et non hachée. Une fois terminé, exécutez le programme pour voir s'il fonctionne.

5.2. Comparaison du nombre de lignes sans tuple et sans instruction *for*

Lorsque les lignes 7 à 13 du [code 5-1-1] sont mises sur une seule ligne, le programme se réduit à 7 lignes de code, comme illustré ci-dessous.

[Exemple de code 5-2-1]

```
from pystubit.board import buzzer
import time

melodie = ('C4', 600),('D4', 500),('E4', 400),('F4', 600),('G4', 500),('A4', 400),('B4', 600),('C5', 500)

for note in melodie:
    buzzer.on(note[0])
    time.sleep_ms(note[1])
buzzer.off()
```

En revanche, si vous n'utilisez ni de tuples ni d'instructions *for*, le programme fera 19 lignes, comme montré ci-dessous. Les tuples et les instructions *for* permettent donc de simplifier un programme et de réduire sa taille.

[Exemple de code 5-2-2]

```
from pystubit.board import buzzer
import time
buzzer.on('C4')
time.sleep_ms(600)
buzzer.on('D4')
time.sleep_ms(500)
buzzer.on('E4')
time.sleep_ms(400)
buzzer.on('F4')
time.sleep_ms(600)
buzzer.on('G4')
time.sleep_ms(500)
buzzer.on('A4')
time.sleep_ms(400)
buzzer.on('B4')
time.sleep_ms(600)
buzzer.on('C5')
time.sleep_ms(500)
buzzer.off()
```

Résumé de ce chapitre

Dans cette leçon, vous avez découvert :
- une nouvelle catégorie de données : les séquences (parmi les types de séquences existantes, vous avez manipulé les listes et les tuples),
- comment écrire des programmes bien faits en utilisant des boucles *while* et *for*.

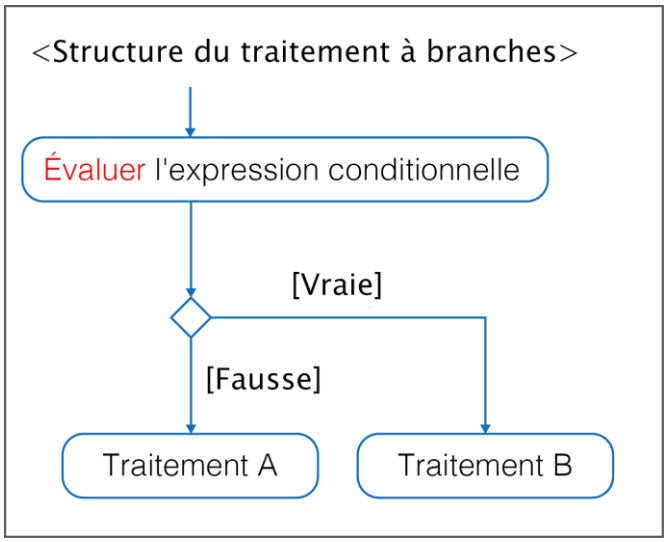
Chapitre 2 : Le traitement à branches

1. Ce que vous allez apprendre dans cette leçon

Au chapitre 1, vous avez découvert les séquences et les boucles. Dans ce chapitre, nous poursuivons notre exploration des traitements de base en abordant cette fois-ci le traitement à branches. Vous apprendrez les bases de ce type de traitement et programmerez les capteurs de l'ESPeRobo pour contrôler le panneau LED.

2. Expressions conditionnelles et valeurs booléennes

Le traitement à branches est une méthode de contrôle qui sépare le traitement en deux branches selon que l'expression conditionnelle est remplie ou non. Avant d'aller plus loin dans le traitement des branches, il convient d'aborder les expressions conditionnelles.



2.1. Les opérateurs de comparaison des expressions conditionnelles et leurs résultats

Comme expliqué pour l'instruction *while* (*cf.* chapitre 1, partie 4.1.), une expression conditionnelle est formulée à l'aide d'opérateurs de comparaison.

Opérateurs de comparaison	Exemples d'utilisation	Signification
==	a == b	a et b sont identiques
!=	a != b	a et b sont différents
<	a < b	a est strictement inférieur à b (exclut le cas a == b)
>	a > b	a est strictement supérieur à b (exclut le cas a == b)
<=	a <= b	a est inférieur ou égal à b (inclut le cas a == b)
>=	a >= b	a est supérieur ou égal à b (inclut le cas a == b)

L'exécution d'une expression conditionnelle formulée avec ces opérateurs de comparaison renvoie un résultat. Observons le type de résultat renvoyé en recopiant, dans le terminal de l'éditeur, le code ci-dessous qui utilise les opérateurs de comparaison == et ! =.

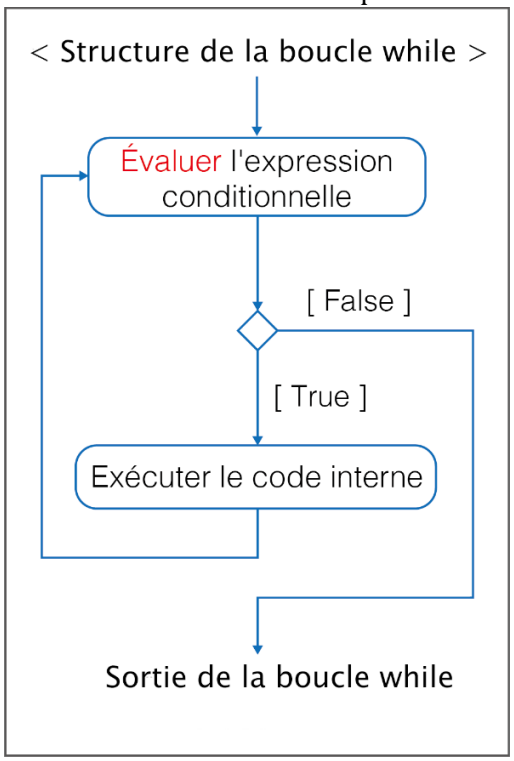
```
>>> 1==1
True
>>> 1!=1
False
```

Ces deux opérations de comparaison renvoient un résultat prenant la forme d'une donnée de type booléen qui n'a que deux valeurs : True ou False. True indique que l'expression conditionnelle est satisfaite. False indique, au contraire, que l'expression conditionnelle n'est pas satisfaite.

Note : Lorsque l'utilisateur emploie directement ces valeurs booléennes, la première lettre de True et de False doit toujours être en majuscule, autrement cela générera une erreur, car le programme considérera que, par exemple, « true » est le nom d'une variable inconnue, à déterminer.

Avec la boucle *while*, le traitement consistait à exécuter à plusieurs reprises le bloc d'instructions interne tant que l'expression conditionnelle était satisfaite et à quitter la boucle quand elle ne l'était plus.

Pour le reformuler en utilisant le type booléen, l'expression conditionnelle est évaluée à l'aide d'une opération de comparaison. Tant que True est renvoyé, alors le bloc d'instructions de la boucle est exécuté de manière répétée. Si False est renvoyé, alors l'itération prend fin.



On peut donc programmer une boucle infinie avec une instruction `while True:` pour répéter indéfiniment son bloc d'instructions.

```
While True :
      Bloc d'instructions de la boucle
      .
      .
      .
```

Si l'on convertit les types booléens True et False en types entiers, ils renverront respectivement « 1 » et « 0 ».

```
>>> int(True)
1

>>> int(False)
0
```

Inversement, si l'on convertit une valeur entière en type booléen, « 0 » se convertira en False et tous les nombres autres que « 0 » se convertiront en True.

```
>>> bool(0)
False

>>> bool(1)
True

>>> bool(10)
True
```

Les chaînes peuvent aussi être converties en booléen : une chaîne vide renverra False, tandis que toute autre chaîne de caractères renverra True.

```
>>> bool('')
False

>>> bool('a')
True

>>> bool('c')
True

>>> bool(' ')
True
```

En mettant « == False », vous pouvez vérifier si le résultat est bien « 0 ». En mettant « == True », vous pouvez vérifier si le résultat est bien 1.

```
>>> 1-1 == False
True

>>> 10-2 == True
False

>>> 10/3 == True
False
```

Pour résumer, E == True renvoie la valeur True si et seulement si le résultat de l'expression E est lui-même le booléen True ou l'entier 1, dans tous les autres cas, la valeur False est renvoyée.

E == False renvoie la valeur True si et seulement si le résultat de l'expression E est lui-même le booléen False ou l'entier 0, dans tous les autres cas, la valeur False est renvoyée.

2.2. Les opérateurs logiques connectant plusieurs expressions conditionnelles

Lors d'un achat, nous prenons généralement des décisions basées sur une combinaison de plusieurs conditions. Par exemple, « Je souhaite acheter des vêtements chauds et abordables » ou encore « Je voudrais des glaces au chocolat ou des biscuits fourrés ». De même, en programmation, un traitement peut combiner plusieurs expressions conditionnelles grâce à un opérateur logique.

Il existe trois opérateurs logiques : « and », « or » et « not ». Parmi eux, « and » et « or » servent à connecter plusieurs expressions conditionnelles.

■ L'opérateur *and*

L'opérateur *and* connecte plusieurs expressions conditionnelles de la façon suivante :

```
Condition 1 and Condition 2
```

L'opérateur *and* ne renvoie True que si les deux expressions conditionnelles sont toutes les deux remplies, sinon il renvoie False.

Résultat de la condition 1	**Résultat de la condition 2**	**Résultat de l'opération *and***
True	**True**	**True**
True	False	False
False	**True**	False
False	False	False

Exécutons les instructions suivantes dans le terminal pour observer leurs résultats.

```
>>> 5 < 10 and 'a' == 'a'
True

>>> 5 < 10 and 'a' == 'A'
False

>>> 5 > 10 and 'a' == 'a'
False

>>> 5 > 10 and 'a' == 'A'
False
```

■ L'opérateur *or*

L'opérateur *or* connecte plusieurs expressions conditionnelles de la façon suivante :

```
Condition 1 or Condition 2
```

L'opérateur *or* renvoie True si le résultat de l'une des deux expressions conditionnelles est remplie et renvoie False uniquement si les deux ne sont pas remplies.

Résultat de la condition 1	**Résultat de la condition 2**	**Résultat de l'opération *or***
True	**True**	**True**
True	False	**True**
False	**True**	**True**

False	False	False

Encore une fois, exécutons le code suivant dans le terminal pour en observer le résultat.

```
>>> 5 < 10 or 'a' == 'a'
True

>>> 5 < 10 or 'a' == 'A'
True

>>> 5 > 10 or 'a' == 'a'
True

>>> 5 > 10 or 'a' == 'A'
False
```

■ L'opérateur *not*

L'opérateur *not* induit la valeur booléenne contraire de l'expression conditionnelle à laquelle il est associé. Autrement dit, *not True* devient *False* et *not False* devient *True*. Exécutons le code suivant dans le terminal pour observer le résultat.

```
>>> not 5 < 10
False

>>> not 5 > 10
True
```

3. Traitement des branches

L'instruction *if-else* est une structure de contrôle conditionnelle qui exécute différentes cellules de code selon que la condition est remplie ou non. L'instruction *if-else* s'écrit de la façon suivante :

```
if condition:
    Code à exécuter si la condition est remplie
else:
    Code à exécuter si la condition n'est pas remplie
```

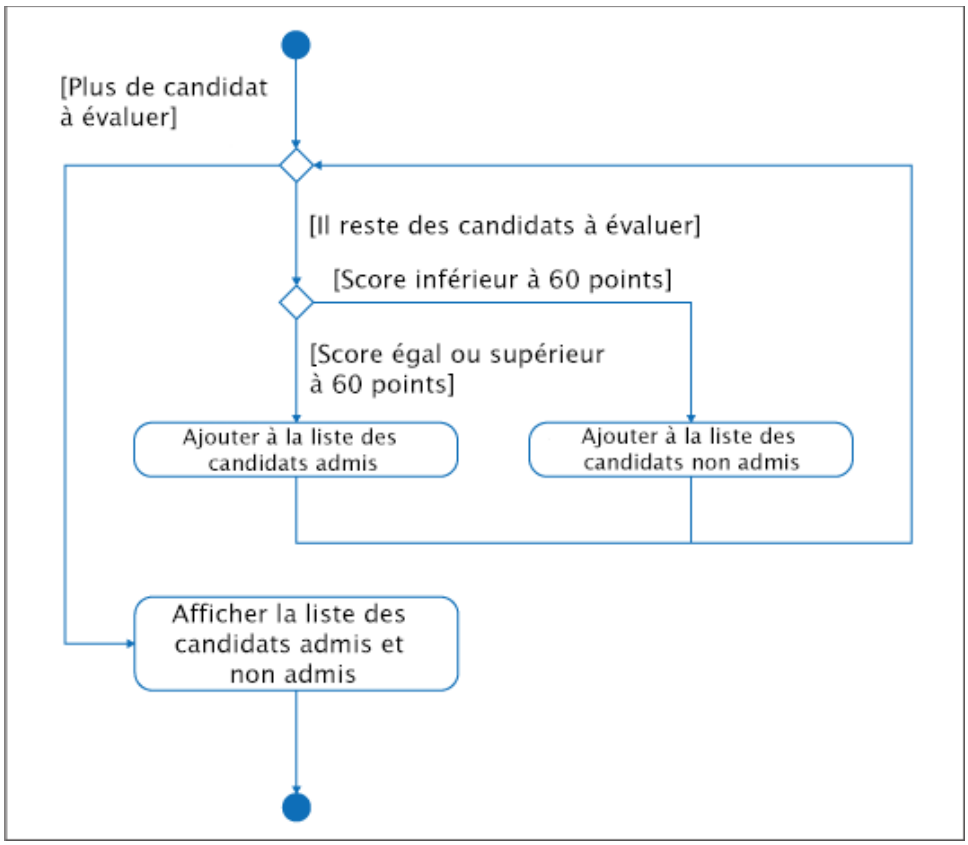
Si vous souhaitez exécuter un processus spécifique uniquement lorsque la condition est remplie, vous pouvez omettre l'instruction *else*. À présent, passons à quelques exercices pratiques pour apprendre à utiliser l'instruction *if-else*.

3.1. Programme de tri entre admis et non admis selon le score obtenu

Les examens et concours permettent de déterminer si des candidats ont atteint le niveau de connaissances et de compétences requis en leur attribuant une note. Lorsque des tests sont passés par de nombreuses personnes, ils peuvent être notés automatiquement par un système qui calcule les bonnes et mauvaises réponses. Nous allons, à présent, écrire un programme qui déterminera la réussite ou l'échec de plusieurs candidats à un examen en fonction du score qu'ils ont obtenu.

■ Analyse du processus du programme à créer

Le programme devra rechercher parmi les candidats ceux qui ont obtenu un score de 60 points ou plus et les ajouter à la liste des candidats admis. Ceux qui ont obtenu un score inférieur seront, quant à eux, ajoutés à la liste des candidats non admis. Une fois ce tri effectué, la liste des admis et des non admis s'affichera.



■ Création du programme

Créons, d'abord, les données relatives aux cinq candidats qui ont passé le test, présentées dans le tableau ci-dessous.

Numéro du candidat	Score
No.01	72
No.02	56
No.03	86
No.04	45
No.05	63

Pour créer ces données (`data`), nous utiliserons un dictionnaire dont la clé sera le numéro du candidat et la valeur, le score obtenu par le candidat en question.

```
1 data = {'No.01':72, 'No.02':56, 'No.03':86, 'No.04':45, 'No.05':63}
```

Créons, ensuite, deux listes vides dans lesquelles seront répartis les candidats admis et non admis.

```
data = {'No.01':72, 'No.02':56, 'No.03':86, 'No.04':45, 'No.05':63}

admis = []
non_admis = []
```

Créons une boucle qui détermine pour chacun des cinq candidats s'ils sont admis ou non. Si vous utilisez une instruction *for ~ in*, vous pourrez traiter les éléments d'un dictionnaire ou d'une liste dans l'ordre. N'oubliez pas, cependant, que pour les dictionnaires, ce sont les clés et non les valeurs qui sont récupérées.

```
for Clé in Dictionnaire:

    Code à exécuter à plusieurs reprises
    .
    .
```

Ici, nous extrayons le numéro du candidat enregistré en tant que clé dans le dictionnaire, la stockons dans la variable « numero » et obtenons sa valeur avec « data[numero] ». Le score du candidat est, ensuite, évalué par l'expression conditionnelle. Le numéro du candidat est ajouté à la liste « admis » si son score est de 60 points ou plus ou ajouté à la liste « non_admis » dans le cas contraire.

```
data = {'No.01':72, 'No.02':56, 'No.03':86, 'No.04':45, 'No.05':63}

admis = []
non_admis = []

for numero in data:
    if data[numero] >=60:
        admis.append(numero)
    else:
        non_admis.append(numero)
```

Affichons, enfin, le contenu des deux listes avec la fonction print(). Plusieurs arguments peuvent être spécifiés dans cette fonction en les séparant par une virgule. Les valeurs de retour s'afficheront sur une seule ligne.

[Exemple de code 3-1-1]

```
data = {'No.01':72, 'No.02':56, 'No.03':86, 'No.04':45, 'No.05':63}

admis = []
non_admis = []

for numero in data:
    if data[numero] >=60:
        admis.append(numero)
    else:
        non_admis.append(numero)

print('admis :', admis)
print('non admis :', non_admis)
```

Appuyez sur le bouton « Lancer » pour exécuter le programme et avoir le résultat.

```
admis : ['No.03', 'No.01', 'No.05']
non admis : ['No.02', 'No.04']
```

■ Comment intégrer plusieurs conditions dans une structure à branches ?

Une structure conditionnelle ne se limite pas forcément à une branche (if) ou deux branches (if-else), mais peut en comporter davantage.

Dans le programme précédent, par exemple, ceux qui avaient obtenu 60 points ou plus étaient admis et ceux qui avaient obtenu moins n'étaient pas admis. Nous pourrions tout à fait enchâsser une autre instruction *if-else* à l'intérieur de l'instruction *if* ou bien à l'intérieur de l'instruction *else*. Reprenons l'exemple du code [3-1-1] pour le modifier dans ce sens.

[Exemple de code 3-2-1]

Addition / modification des [lignes 5, 11 à 14 et 18]

```
1  data = {'No.01':72, 'No.02':56, 'No.03':86, 'No.04':45, 'No.05':63}
2
3  admis = []
4  non_admis = []
5  rattrapage = []
6
7  for numero in data:
8      if data[numero] >=60:
9          admis.append(numero)
10     else:
11         if data[numero] >= 55:
12             rattrapage.append(numero)
13         else:
14             non_admis.append(numero)
15
16
17 print('admis :', admis)
18 print('non admis :', non_admis)
19 print('rattrapage :', rattrapage)
```

(Résultat après exécution)

```
admis : ['No.03', 'No.01', 'No.05']

non admis : ['No.04']

rattrapage : ['No.02']
```

Dans le programme ci-dessus, le premier « `if data[numero]>=60:` » sert à vérifier si le score est supérieur ou égal à 60 points, le deuxième sert, quant à lui, à vérifier si le score est de 55 points ou plus.

Outre l'enchâssement *if-else*, vous pouvez aussi utiliser *elif* (sinon si). Une ou plusieurs autres expressions conditionnelles *elif* peuvent être insérées un nombre illimité de fois entre l'instruction *if* et l'instruction *else*.

```
if condition 1 :
            Code exécuté lorsque la condition 1 est remplie
            .
            .
elif condition 2 :
            Code à exécuter quand la condition 1 n'est pas remplie et
que la condition 2 est remplie
            .
            .
elif condition 3 :
            Code à exécuter quand les conditions 1 et 2 ne sont pas
remplies et que la condition 3 est remplie
            .
            .
else :
           Code à exécuter quand aucune des conditions n'est
           remplie
           .
           .
```

Le [code 3-2-1] peut être réécrit de la façon suivante avec une instruction **elif**.

[Exemple de code 3-2-2]
Modifier [lignes 11-14]

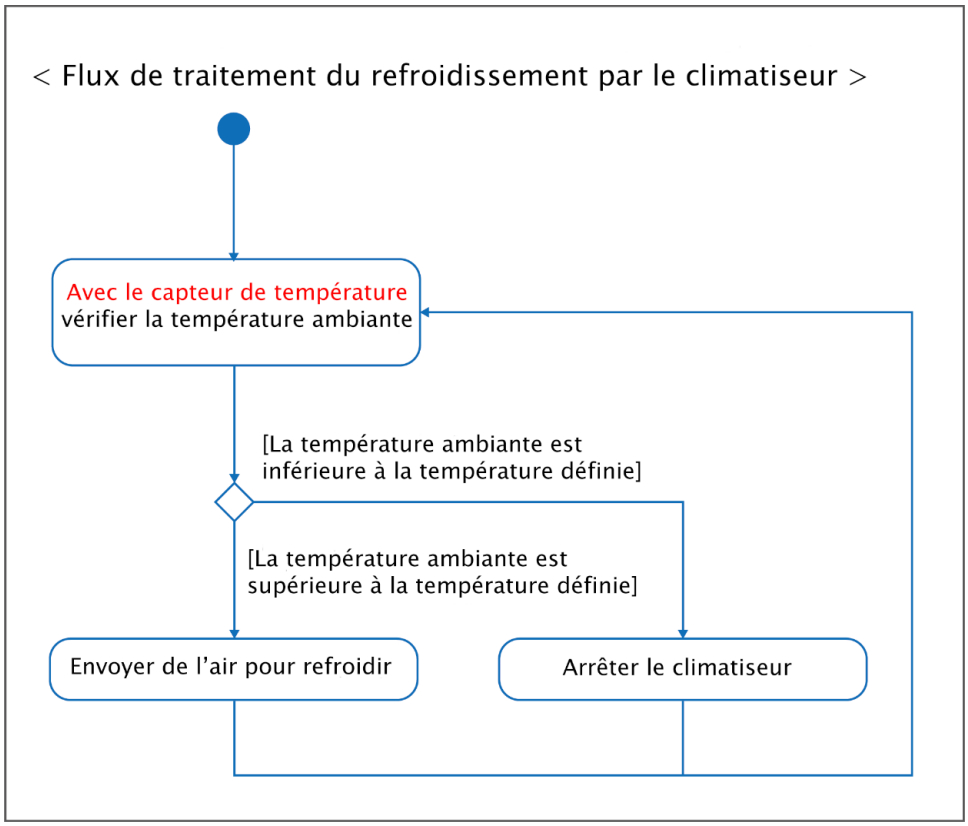
```
1  data = {'No.01':72, 'No.02':56, 'No.03':86, 'No.04':45, 'No.05':63}
2
3  admis = []
4  non_admis = []
5  rattrapage = []
6
7  for numero in data:
8      if data[numero] >=60:
9          admis.append(numero)
10     elif data[numero] >=55:
11         rattrapage.append(numero)
12     else:
13         non_admis.append(numero)
14
15
16 print('admis :', admis)
17 print('non admis :', non_admis)
18 print('rattrapage :', rattrapage)
```

Comme l'ont montré ces deux exemples, il n'y a pas une façon unique d'écrire un programme pour obtenir le même résultat. Cependant, pour éviter les confusions entre différentes méthodes potentielles pour écrire un programme, il est recommandé d'établir au préalable un organigramme pour organiser le flux du traitement avant de se lancer dans l'écriture du programme lui-même.

4. Défi : faire fonctionner automatiquement des objets avec des capteurs

Les capteurs, comme les sens humains (la vue, l'ouïe, le toucher, le goût et l'odorat), collectent des informations sur leur environnement. De nombreux appareils électroménagers et robots utilisent ces capteurs pour fonctionner automatiquement.

Par exemple, le climatiseur modifie la température ambiante d'une pièce pour qu'elle corresponde à la température programmée en s'aidant d'un capteur de température.



Certains modèles de climatiseurs sont maintenant équipés de nouveaux capteurs. Ils peuvent être, par exemple, dotés d'un capteur qui permet, selon les fonctionnalités proposées, de détecter les personnes pour envoyer de l'air vers elles ou pour arrêter le climatiseur s'il n'y a personne.

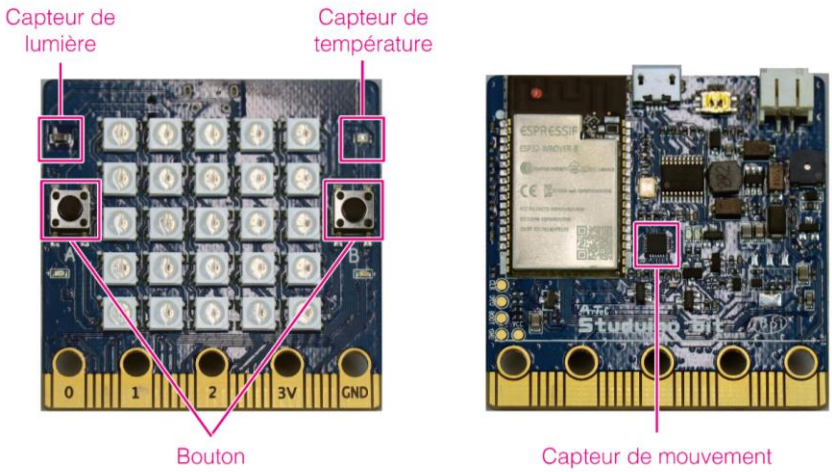
À la fin de la leçon, nous allons créer un programme qui utilise le capteur intégré de l'ESPeRobo pour séparer le traitement en fonction des informations environnantes recueillies.

4.1. Les capteurs intégrés de l'ESPeRobo

L'ESPeRobo comporte quatre capteurs : un capteur de lumière, un capteur de température, deux boutons et un capteur de mouvement. Dans cette leçon, nous utiliserons le capteur de lumière.

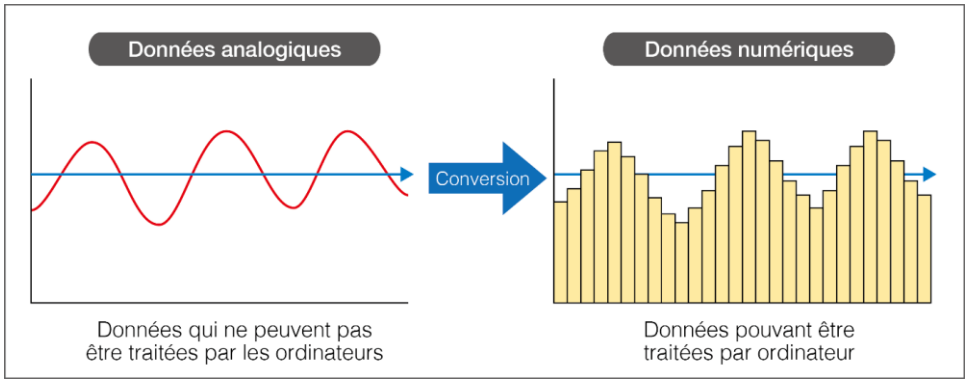
Nom du capteur	**Rôle**
Capteur de lumière	Vérifier la luminosité environnante
Capteur de température	Vérifier la température ambiante

Bouton	Vérifier si le bouton est pressé
Capteur de mouvement	Mesurer l'accélération (avec l'accéléromètre), la vitesse angulaire (avec le gyroscope) et le magnétisme (avec la boussole)



4.2. Observer la luminosité environnante avec le capteur de lumière

Les données récupérées sur la luminosité et la température sont des données analogiques - c'est-à-dire des données représentées sous la forme de variations continues d'une grandeur physique - qui ne peuvent donc pas être traitées telles quelles par un ordinateur. C'est pourquoi les données analogiques recueillies par le capteur sont converties en données numériques qui, elles, peuvent être traitées par un ordinateur. Les données obtenues par les capteurs intégrés de l'ESPeRobo sont donc toutes représentées par des valeurs numériques.



À présent, écrivons un programme pour voir quel type de données numériques s'affiche pour mesurer la luminosité environnante lorsqu'on utilise un capteur de lumière.

■ Écriture du programme

Recopiez, d'abord, les deux lignes de code suivantes dans la zone d'édition.

```
1  from pystubit.board import lightsensor
2  import time
```

La première ligne de code appelle l'objet « lightsensor » contenu dans le module pystubit.board pour récupérer les données du capteur de lumière. Pour rappel, un module en Python est une sorte de bibliothèque d'objets (fonctions, constantes, etc.) dédiés à une tâche particulière. La deuxième ligne de code appelle, quant à elle, l'objet « time » pour contrôler le temps dans le programme.

Écrivons maintenant un code qui récupère toutes les 500 millisecondes les données du capteur de lumière. Utilisons, pour cela, la boucle « `while True:` » présentée plus tôt dans ce chapitre.

Ajouter la [ligne 4]

```
1  from pystubit.board import lightsensor
2  import time
3
4  while True:
```

Pour obtenir la luminosité, utilisez la méthode get_values() avec l'objet lightsensor. Cette méthode renvoie une valeur entière représentant la luminosité. Stockons cette valeur de retour dans la variable « value » et affichons-la dans le terminal avec la fonction print(). Ajoutez, enfin, à la dernière ligne, une méthode sleep_ms() pour l'objet time afin que les données du capteur soient récupérées toutes les 500 millisecondes.

[Exemple de code 4-2-1]

Ajouter les [lignes 5-7]

```
1  from pystubit.board import lightsensor
2  import time
3
4  while True:
5      value = lightsensor.get_value()
6      print(value)
7      time.sleep_ms(500)
```

*Attention ! Si le temps spécifié par time.sleep_ms(500) à la ligne 7 est trop court, le traitement d'affichage lors de l'exécution de la fonction print() sera lourd et le logiciel Mu pourrait ne plus fonctionner.

■ Exécution du programme

Exécutez-le [code 4-2-1] pour voir comment les valeurs affichées changent entre le moment où le capteur de lumière est exposé et celui où il est recouvert par votre main.

(Exemple d'affichage lorsque le capteur de lumière est exposé)

```
4095
4095
4095
4095
4095
```

(Exemple d'affichage lorsque le capteur de lumière est recouvert)

```
871
722
693
683
684
```

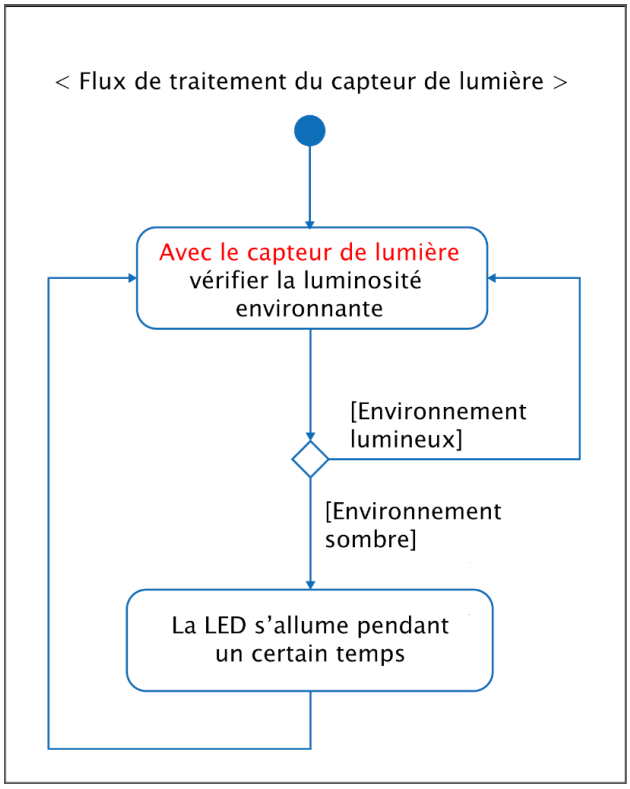
La luminosité ambiante recueillie par le capteur de lumière est représentée par une valeur entière comprise dans une plage allant de 0 à 4095. Plus la valeur est faible, plus la luminosité est faible, et plus la valeur est élevée, plus la luminosité est élevée.

Diviser le traitement du programme en fonction de la luminosité ambiante

Écrivons un programme qui utilise la luminosité ambiante recueillie par le capteur de lumière pour séparer le traitement du programme. À titre d'exemple, imaginons ici un capteur de lumière qui s'allume automatiquement lorsque l'environnement s'assombrit. Écrivons le code de façon à ce que le panneau LED s'allume pendant un certain temps lorsque la valeur du capteur de lumière tombe en dessous d'une certaine valeur.

■ Le flux de traitement du programme

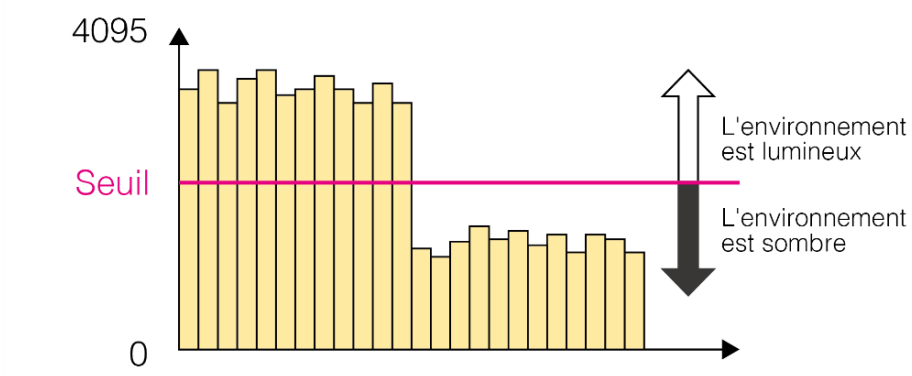
L'organigramme ci-dessous montre le flux de traitement du programme. Comme on peut le voir, le programme examine constamment la luminosité environnante pour pouvoir réagir immédiatement lorsque la luminosité change et allumer le panneau LED.



■ Le réglage du seuil

Pour ce programme, il faut déterminer le moment à partir duquel l'ordinateur estimera que l'environnement est sombre.

Comme la luminosité est convertie et traitée en données numériques par l'ordinateur, il suffit de comparer les données numériques obtenues quand l'environnement est sombre et éclairé, pour déterminer une valeur de référence qui fera office de limite entre ces deux états.



La valeur de référence pour séparer un certain état d'un autre est appelée « seuil ». À partir de vos observations, déterminez la valeur numérique à partir de laquelle vous souhaitez que le panneau LED s'allume.

■ Création du programme

Continuons à écrire le programme en utilisant le seuil qui a été choisi.

Ajoutons, tout d'abord, le code qui nous permettra d'utiliser le panneau LED de l'ESPeRobo. Pour cela, ajoutez dans votre programme les lignes 2, 3 et 6 comme dans le programme montré ci-dessous.

Ajouter les [lignes 2, 3 et 6]

```
1  from pystubit.board import lightsensor
2  from pystubit.board import display
3  from pystubit.board import Image
4  import time
5
6  image = Image('11111:11111:11111:11111:11111:')
7
8  while True:
9      value = lightsensor.get_value()
10     print(value)
11     time.sleep_ms(500)
```

À la ligne 6, entre les parenthèses de l'objet Image, se trouvent des informations sur la façon dont va s'éclairer le panneau LED. Dans ce code, les 5 LED des 5 rangées du panneau s'allumeront.

Ajoutons maintenant, dans l'instruction *while*, une structure conditionnelle. Recopiez le code, mais n'oubliez pas de remplacer, dans la condition de l'instruction *if* à la ligne 10, la valeur « 1000 » par le seuil que vous avez choisi.

[Exemple de code 4-3-1]

Addition / modification des [lignes 10 à 13]

```
1  from pystubit.board import lightsensor
2  from pystubit.board import display
3  from pystubit.board import Image
4  import time
5
6  image = Image('11111:11111:11111:11111:11111:')
7
8  while True:
9      value = lightsensor.get_value()
10     if value < 1000:
11         display.show(image)
12         time.sleep_ms(1000)
13         display.clear()
```

À la ligne 10 du programme, le processus s'exécute si la valeur recueillie par le capteur est inférieure au seuil.

Le code de la ligne 11, `display.show(image)`, allume les LED selon le motif défini à la ligne 6 du programme dans l'objet Image grâce à la méthode show(). La méthode show(), qui contrôle l'affichage du panneau LED comme la méthode scroll(), permet d'afficher un motif en commutant une à une les LED des 5 rangées du panneau.

À la ligne 13, la méthode clear() de l'objet display permet d'éteindre toutes les LED allumées.

■ Exécution du programme

Exécutez le [code 4-3-1] et vérifiez que l'affichage LED s’allume en rouge lorsque vous recouvrez le capteur de lumière avec votre main.

Résumé de cette leçon

Dans cette leçon, vous avez découvert les traitements à branches et appris à écrire des conditions.

Parmi les notions acquises, vous avez appris à :
- créer des conditions à l'aide d'opérateurs de comparaison.
- connecter plusieurs conditions avec des opérateurs logiques ("and" et "or").

Vous avez appris les notions suivantes :
- Le résultat d'une opération de comparaison doit être une valeur booléenne ("True" ou "False").
- Avec une instruction *if-else*, le traitement peut être différencié selon le résultat de l'évaluation d'une condition.
- Un traitement peut comprendre plus de deux branches en imbriquant une autre instruction *if-else* ou des instructions *elif*.

Avec les traitements itératifs et à branches, vous pourrez désormais réaliser des opérations complexes que ne pourrait pas faire le traitement séquentiel.

Vous avez également appris à utiliser le capteur de lumière de l'ESPeRobo. Les autres capteurs intégrés de l'ESPeRobo seront abordés dans les leçons suivantes.

Ecole Algora

Apprendre à coder en s'amusant

PLUS DE 36 ROBOTS DIFFÉRENTS
UN CURSUS COMPLET
POUR APPRENDRE À PROGRAMMER

2 Cursus
6-9 ans
9-14 ans

Inscris-toi sur www.algora.school

Apprendre à programmer des robots pour comprendre le monde d'aujourd'hui et de demain.

Les machines programmées, de plus en plus intelligentes, font partie intégrante de notre vie de tous les jours. Elles nous accompagnent, nous entourent et ont envahi tous les domaines de notre vie quotidienne. Maîtriser le monde, ce n'est pas les utiliser, mais avant tout comprendre comment elles fonctionnent.

Comment fonctionnent-elles ?
Selon quelle logique ? Selon quels algorithmes ?
Comment sont conçus les programmes qui leur dictent leurs actions et réactions ?

C'est ce que vous apprendrez tout au long de ces livrets d'apprentissage. Et pas seulement "en théorie" : vous allez vous-même concevoir et programmer vos propres robots : des actions simples aux plus complexes, vous apprendrez à programmer des robots amusants et originaux que vous aurez conçus vous-même. Une seule limite : votre créativité !

École Robots permet à tous de s'initier à la programmation en s'amusant, un enjeu majeur, aujourd'hui et demain.

Ecole Robots
Apprendre à coder avec les robots éducatifs
CURSUS EDUCATION NATIONALE

Pour en savoir plus : www.ecolerobots.com

Réf : LIV-EN-LY-PY-2(v.1)