1. Présentation de l'ARDUINO UNO

1.1. ARDUINO UNO, Qu'est-ce que c'est ?

1.2. Découverte de la carte ARDUINO UNO

1.3. Le logiciel "ARDUINO IDE" - Initiation au langage ARDUINO

1.4. Les bases de la programmation

1.4.1. Structure du programme

1.4.1.1. Syntaxe de base

1.4.1.2. Les opérateurs arithmétiques

1.4.1.3. Les opérateurs de comparaison (==, !=, <, >)

1.4.1.4. Les opérateurs booléens

1.4.1.5. Les opérateurs composés

1.4.1.6. Structures de contrôle

1.4.2. Variables et constantes

1.4.2.1. Les constantes Arduino prédéfinies

1.4.2.2. Les variables – Types de données

1.4.3. Les fonctions

1.4.3.1. Fonctions des Entrées/Sorties numériques

1.4.3.2. Fonctions des Entrées/Sorties analogiques

1.4.3.3. Fonctions des Entrées/Sorties avancées

1.4.3.4. Fonctions de gestion du temps

1.4.3.5. Fonctions de communication (librairie Serial)

1.4.3.6. Fonctions propres au programme

1.1. ARDUINO UNO, Qu'est-ce que c’est

Une équipe de développeurs composée de Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, David Mellis et Nicholas Zambetti a imaginé un projet répondant au doux nom de ARDUINO UNO et mettant en œuvre une petite carte électronique programmable et un logiciel multiplateforme, qui puisse être accessible à tout un chacun, dans le but de créer facilement des systèmes électroniques.

L’ARDUINO UNO fait partie de la famille des platines de développement. Contrairement au Raspberry Pi et aux Beaglebone, il ne possède pas de système d’exploitation basé sur Linux. Il reste par contre l'un des plus abordables et des plus répandus.

Les platines de développement (ou les microcontrôleurs) sont des circuits intégrés regroupant plusieurs éléments :

- un microprocesseur,
- de la mémoire de type RAM,
- de la mémoire non volatile pour stocker un microprogramme (firmware),
- des ports de communication de type USB, I2C, Ethernet, Bluetooth, wifi….
- des convertisseurs analogique-numériques (CAN) …

Un microcontrôleur comme son nom l’indique, permet de contrôler des composants électroniques et mécaniques : capteurs, lumières, moteurs, vannes, etc…, afin de créer des systèmes automatisés et régulés, comme par exemple, un système d’arrosage automatique, la régulation de température d’une pièce, etc…

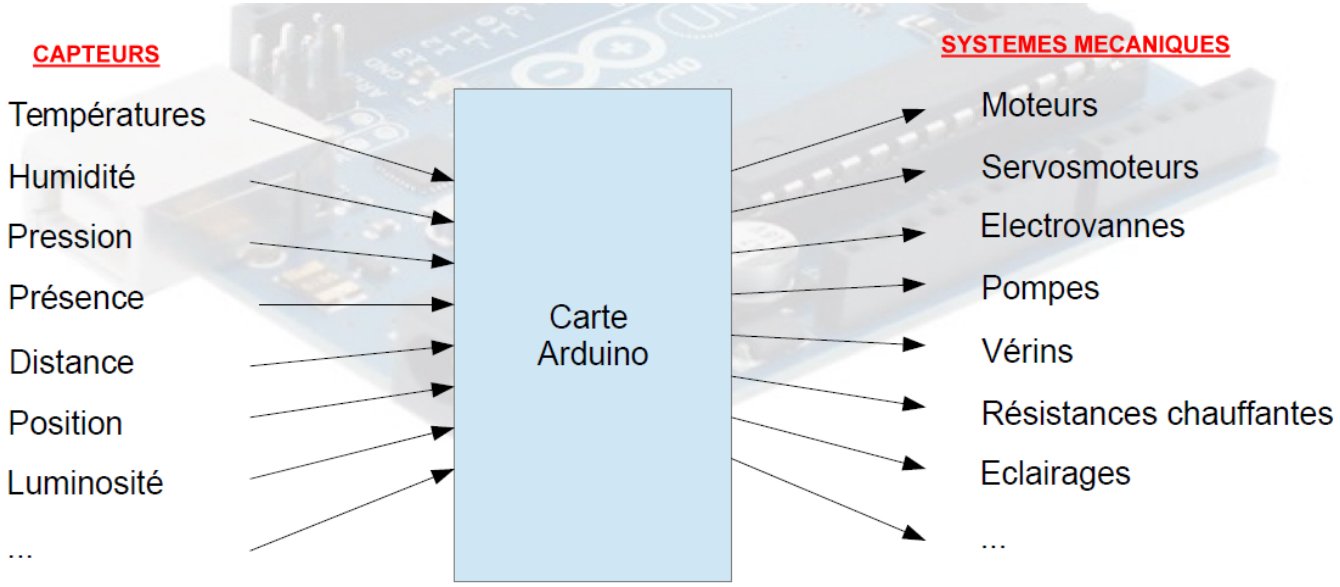
Les microcontrôleurs sont présents dans de nombreuses applications :

- Électronique embarquée (voiture, avion …)
- instruments de mesures médicaux, organes artificiels,

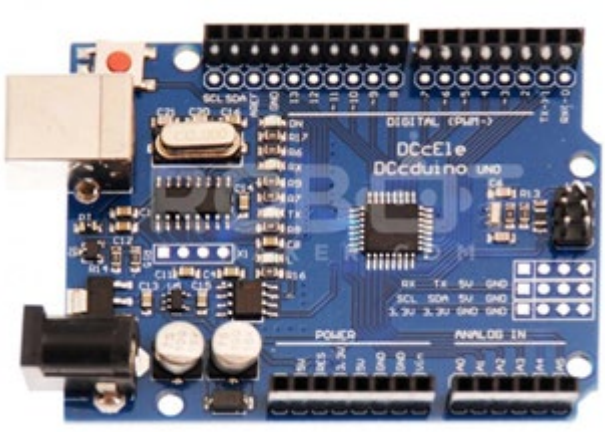
- objets de la sécurité : alarme, caméra de surveillance …
- les appareils électriques du quotidien : machine à laver, hifi …

L'Arduino Uno est donc un microcontrôleur programmable et éventuellement contrôlables avec un ordinateur.

Généralement, dans un système automatisé avec un Arduino Uno, les données d'un capteur (température, humidité, luminosité, etc…) sont transmises au microcontrôleur qui, en fonction des valeurs reçues, donnent l'ordre à un système mécanique d'effectuer une action (allumer ou éteindre une lampe, ouvrir ou fermer une vanne, etc…)

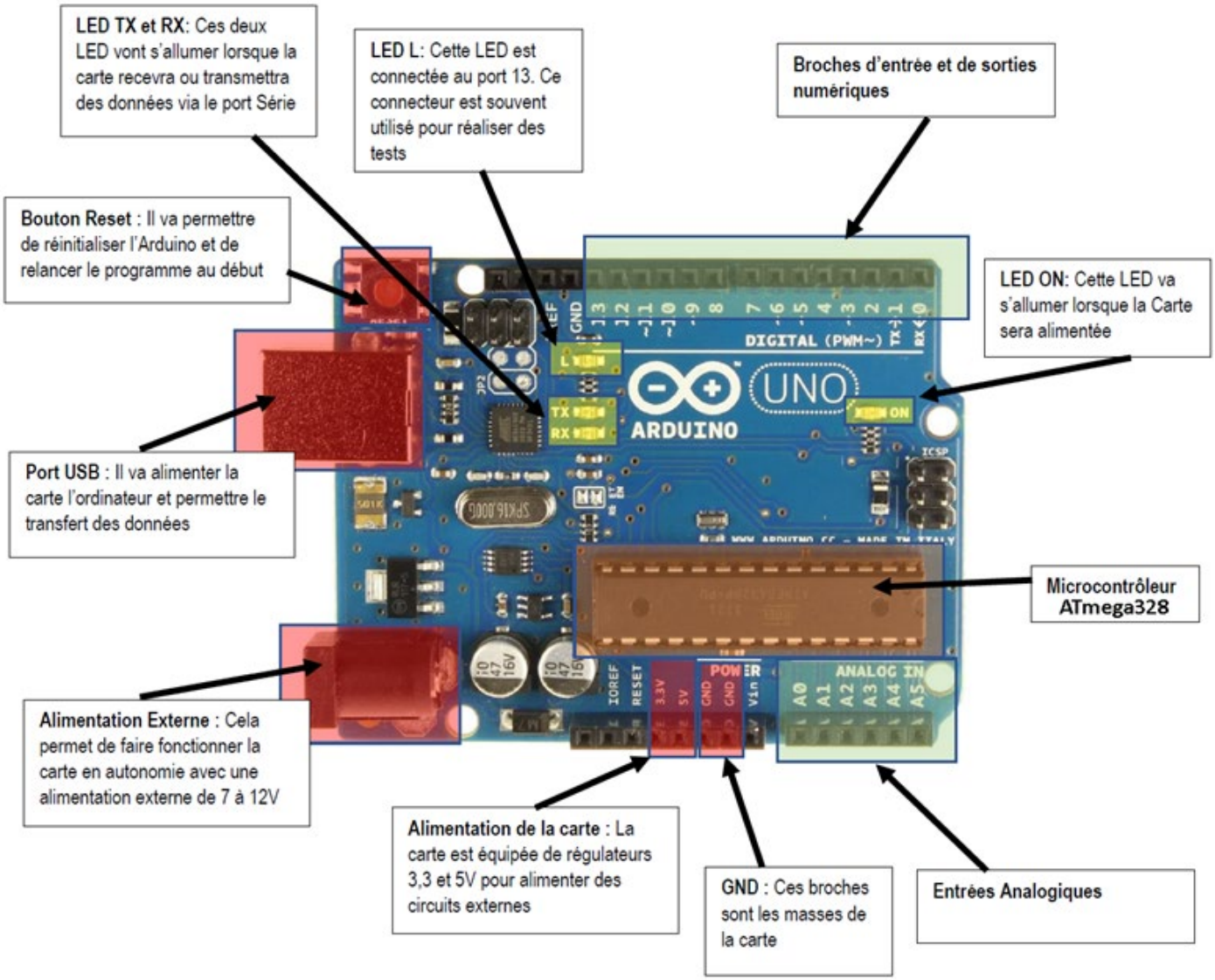


Comme l'Arduino UNO est open source, il existe un grand nombre de clones et de platines compatibles :



1.2. Découverte de la carte ARDUINO

. Schéma d'une platine Arduino Uno



. Le microcontrôleur

La puce ATmega328 (Microcontrôleur) est le cerveau de l'Arduino. C'est un microcontrôleur de la série AVR produite par la société Atmel (16 MHz d'horloge et 32 Ko de mémoire).

Il va recevoir le programme que nous allons créer et va le stocker dans sa mémoire avant de l'exécuter. Grâce à ce programme, il va savoir faire des choses, qui peuvent être : faire clignoter une LED, afficher des caractères sur un écran, envoyer des données à un ordinateur, mettre en route ou arrêter un moteur…

Il existe deux modèles d'Arduino Uno. L'un avec un microcontrôleur de grande taille, et un autre avec un microcontrôleur dit SMD (SMD : *Surface Mounted Device*, soit composants montés en surface, en opposition aux composants qui traversent la carte électronique et qui

sont soudés du côté opposé). D'un point de vue utilisation, il n'y a pas de différence entre les deux types de microcontrôleurs.

. Les mémoires

- Mémoire flash
Elle est de 32 ko dont 0,5 ko sont utilisés pour le *boot loader*, son programme de démarrage. Cette mémoire est l'équivalent du disque dur pour l'ordinateur. C'est la place que nous avons pour stocker le programme.

- SRAM
Équivalent à la mémoire RAM, elle sert à stocker le résultat des variables. Sa taille est de 2 ko. Comme la RAM, cette mémoire est volatile, à l'extinction de la carte, les valeurs disparaissent.

- EEPROM
Mémoire en dur, elle permet de sauvegarder des valeurs de variables et ceci même à l'extinction de la carte. Sa taille est de 1 kB. Comme une carte SD, le nombre de réécritures est limité.

Ces mémoires sont limitées en taille. Il s'agit donc d'optimiser au maximum afin que la carte Arduino Uno puisse recevoir le programme et l'exécuter correctement.

. Le transfert des données

L'Arduino dispose de 3 voies de communication avec les différents éléments du système qu'il contrôle :

. Le port USB

La connexion de la carte Arduino à l'ordinateur se fait par le connecteur USB qui établit une liaison série (**COM**) pour le transfert des données entre l'Arduino et l'ordinateur

C'est évidemment le connecteur USB que l'on utilise lorsqu'on veut transférer un programme écrit avec le logiciel "**IDE ARDUINO**" pour faire fonctionner un montage réalisé avec la carte.

Quand l'Arduino est connecté à un ordinateur, Le connecteur USB sert également d'alimentation en +5 V de la carte.

Attention :

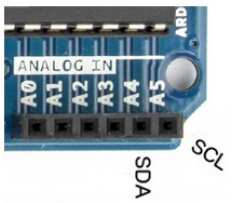
Il est conseillé de débrancher le connecteur USB et toute alimentation externe avant toute manipulation sur le montage avec l'Arduino.

. Le bus I2C

Le bus I2C est l'un des bus de communication disponibles pour dialoguer soit avec un autre Arduino, soit avec des circuits périphériques disposant de ce bus. 2 broches sont employées pour l'I2C, une pour l'horloge et l'autre pour le transfert des données.

De nombreux circuits périphériques disposent d'une interface I2C. On peut citer les expanseurs d'entrées/sorties, circuits qui permettent d'ajouter des entrées/sorties numériques à l'Arduino et les générateurs de PWM qui permettent d'augmenter le nombre de PWM pilotables.

En ce qui concerne l'architecture matérielle, le câblage est très simple. On utilise les connecteurs A4 pour SDA (les données) et A5 pour SCL (l'horloge).



. Le bus SPI

Le bus SPI est également un bus de communication destiné à dialoguer avec les circuits périphériques. À la différence de l'I2C, il est bidirectionnel. Il permet aussi un débit de communication plus important. 4 broches sont employées pour le SPI :

- les données en sortie : **MOSI** - Master out, Slave In (cette broche transporte les données de l'Arduino vers le ou les périphériques SPI) ;

- les données en entrée : **MISO** - Master in, Slave out (cette broche transporte les données du ou des périphériques SPI vers l'Arduino) ;

- la sélection du circuit avec lequel l'Arduino veut dialoguer : **SS** - Slave select (cette broche permet de signifier au périphérique connecté sur quel bus nous souhaitons communiquer avec lui. Chaque périphérique SPI dispose de sa broche SS connectée à l'Arduino) ;

- l'horloge : SCK - Serial clock.

La gestion d'une liaison SPI avec l'Arduino est facile à mettre en œuvre. L'Arduino dispose d'une interface SPI dont les entrées/sorties se répartissent comme suit : **SCK** sur 13, **MISO** sur 12, **MOSI** sur 11 et **SS** sur 10.

. Les entrées et sorties numériques

Il y a 14 entrées/sorties numériques notées de 0 à 13 sur l'Arduino Uno. Elles sont situées sur la grande rangée du haut de la carte. Ces connecteurs (ou broches) sont numériques car le signal sur ces broches ne connaît que deux états (niveau logique) : haut ou bas (1 ou 0). Électriquement, cela se traduit, respectivement, par une tension de 5 V ou 0 V.

Ces broches sont configurées en entrées ou sorties numériques par programmation :

. Configurées en sortie, elles ne peuvent délivrer que des niveaux logiques bas (0 V) et des niveaux logiques haut (5 V).

. Configurées en entrée, elles ne peuvent recevoir que des niveaux logiques bas (0 V) et niveaux logiques haut (5 V).

Attention :

. Les tensions appliquées sur les broches d'entrées/sorties numériques doivent être comprise entre 0 et 5 V. Au-dehors de ces limites, le microcontrôleur sera endommagé.

. Pour une broche configurée en entrée, toute tension inférieure à **0,3 x Vcc**, Vcc étant égale à 5V, soit **1,5 V**, sera comprise comme un niveau logique bas (0 V) et toute tension supérieure à **0,6 x Vcc**, soit **3 V**, sera comprise comme un niveau logique haut (5 V).

Entre les deux, c'est incertain. L'Arduino renverra de toutes façons un 0 ou un 1 mais de manière plus ou moins aléatoire.

. Pour une broche configurée en sortie, il est préférable de limiter l'intensité du courant dans le circuit électrique à **20 mA** et absolument nécessaire de ne pas dépasser **40 mA** sous peine de destruction de la sortie. On veillera aussi à ne pas dépasser une intensité totale de **200 mA** dans les circuits électriques reliés à ces broches.

. Il est déconseillé d'utiliser les broches 0 (RX) et 1 (TX) qui sont initialement prévues pour la communication série (broche 1 pour l'émission et broche 0 pour la réception des données) notamment avec certains modules externes (par exemple, le module Bluetooth HC-05). Mais, en l'absence de communication série, il est possible de les utiliser classiquement en entrée ou sortie numérique.

. Les entrées analogiques

Il y a six entrées analogiques notées de A0 à A5 en bas à droite de la carte.

Les tensions, entre 0 et 5V, appliquées sur ces broches, sont numérisées via un convertisseur analogique-numérique CAN ou ADC (Analog Digital Converter).

Le convertisseur des Arduino effectue une conversion sur 10 bits, c'est à dire qu'il convertit la tension en un nombre entier ayant une valeur de 0 à 1023.

0 correspond à une tension de 0 V et 1023 à une tension de 5V.

La résolution, c'est à dire la différence entre deux valeurs successives de la tension correspondant à une différence de 1 sur l'entier résultat de la conversion analogique-numérique, est donc d'environ 5mV (**5/1024 = 4,9 mV**).

Attention :

. Appliquer une tension supérieure à 5 volts ou inférieure à 0 volt sur une broche analogique endommagera immédiatement et définitivement votre carte Arduino,

. La mesure prend environ 100µs, cela fait un maximum de 10 000 mesures par seconde,

. Effectuer une mesure d'une tension sur une broche non connectée retourne des valeurs de l'ordre de 300 à 500, même s'il n'y a pas de signal,

. La précision de la mesure (conversion sur 10 bits) n'est pas modifiable. Celle-ci est de plus ou moins 1 (+ ou - 5 mV).

Remarque :

Les broches notées de A0 à A5 peuvent également être configurées en entrées et sorties numériques.

. Les sorties analogiques (PWM)

Il n'y a pas de sortie analogique à proprement parler sur un Arduino. Par contre, six des connecteurs numériques (les connecteurs 3, 5, 6, 9, 10 et 11) sont capables de simuler des sorties analogiques et fonctionnent donc comme telles pour délivrer une tension entre 0 et 5 V.

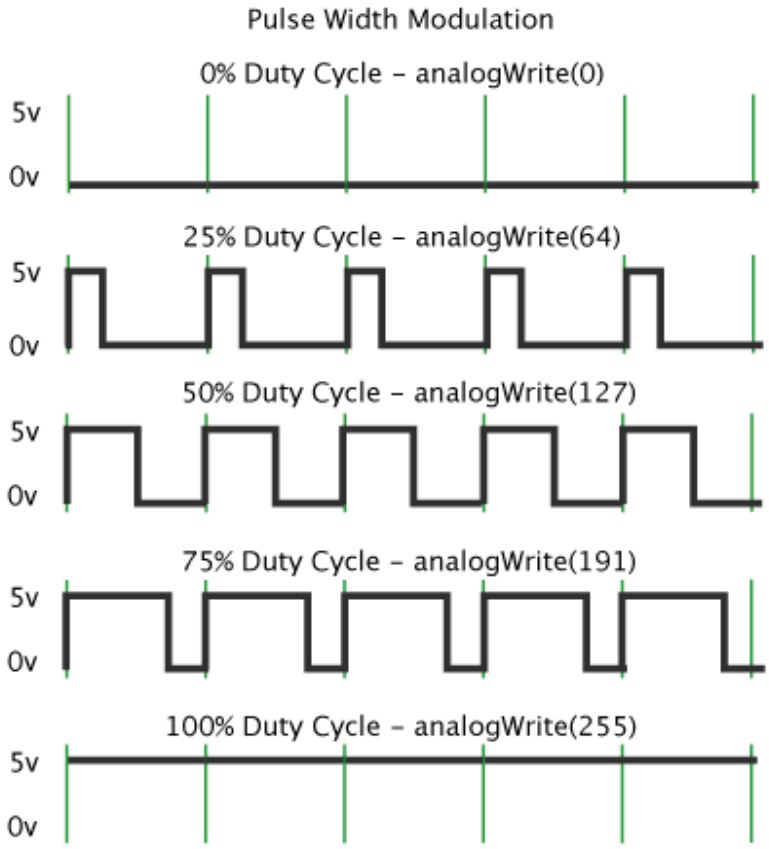
Elles sont marquées du symbole tilde ~ et du sigle PWM, qui veut dire Pulse Width Modulation (Modulation de Largeur d'Impulsion en français).

Le PWM fonctionne ainsi : comme il n'est possible que d'envoyer des informations binaires (haut ou bas, c'est à dire 5 V ou 0 V) sur ces broches, l'Arduino va faire varier la durée pendant laquelle ces deux valeurs sont appliquées afin d'obtenir la tension souhaitée.

L'Arduino génère donc un signal carré caractérisé par deux paramètres (Cf. ci-dessous) :

. L'amplitude du signal est de 5V ou de 0V,

. Le rapport entre la durée où la tension est à 5V et celle où elle est à 0V (ce rapport est appelé Duty cycle et est exprimé en %).



La fréquence du signal PWM est de 490 Hz, ce qui est suffisamment rapide pour que l'on puisse dire que l'amplitude du signal d'une sortie PWM est égale à la valeur moyenne du signal carré généré :

Tension sortie analogique (en V) = 5 x (Duty Cycle/100)

Exemple :

Duty Cycle à 0% : Tension sortie analogique = 5 x 0 = 0 V

Duty Cycle à 25% : Tension sortie analogique = 5 x 0,25 = 1,25 V

Duty Cycle à 50% : Tension sortie analogique = 5 x 0,5 = 2,5 V

Duty Cycle à 75% : Tension sortie analogique = 5 x 0,75 = 3,75 V

Duty Cycle à 100% : Tension sortie analogique = 5 x 1 = 5 V

Remarque :

En langage ARDUINO IDE, la fonction "**analogWrite ()**" permet de générer un signal analogique sur une sortie PWM.

Elle prend deux arguments :

. Le premier est le numéro de la broche sur laquelle on veut générer la PWM
. Le second argument représente la valeur du rapport cyclique à appliquer.

Cependant, on n'exprime pas cette valeur en pourcentage, mais avec un nombre entier compris entre 0 et 255.

Le rapport cyclique s'exprime de 0 à 100 % en temps normal. Cependant, dans cette fonction il s'exprimera de 0 à 255 (sur 8 bits). Ainsi, pour un rapport cyclique de 0% nous enverrons la valeur 0, pour un rapport de 50% on enverra 127 et pour 100% ce sera 255. Les autres valeurs sont bien entendu considérées de manière proportionnelle entre les deux.

. Les broches d'alimentation

- broche "3,3 V" : L'Arduino est équipé d'un régulateur 3,3V dont la sortie est connectée à cette broche. On peut s'en servir pour alimenter un circuit externe en 3,3V,

- **broche "5 V"** : L'Arduino est également équipé d'un régulateur 5V. La sortie de ce régulateur est connectée sur cette broche qui peut donc être employée pour alimenter des circuits externes en 5V. Si on désire construire sa propre alimentation 5V, on peut également fournir cette tension à l'Arduino via cette broche,

- les **broches "GND"**, qui signifie "ground" ("terre", en français), servent à relier le circuit externe à la masse, c'est à dire au 0 V,

- la **broche "Vin"** est l'entrée du régulateur de tension de l'Arduino. On peut choisir d'alimenter l'Arduino via cette broche au lieu d'utiliser la prise d'alimentation,

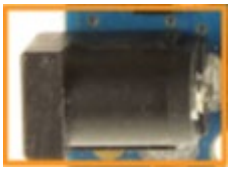
- **broche "IOREF"** : Cette broche est destinée à indiquer aux shields (cartes d'extension avec des fonctions diverses qui s'enfichent sur la carte Arduino) la tension de fonctionnement de l'Arduino. Sur un Arduino 5V, elle aura une tension de 5V et sur une platine en 3,3V, une tension de 3,3 V,

- **broche "RESET"** : Cette broche permet de réinitialiser l'Arduino et donc de redémarrer le programme au début. Pour réinitialiser l'Arduino, il suffit de mettre cette broche à 0 V puis de la repasser à 5 V, ou à 3,3 V selon la platine,

- **broche "AREF"** : La tension appliquée directement à la broche AREF gouverne le module de conversion analogique-numérique du microcontrôleur. Celui-ci renvoie la valeur maximale, 1023, pour cette tension. C'est alors la tension de référence pour les broches analogiques.

. Le connecteur jack d'alimentation supplémentaire

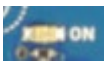
Le connecteur jack d'alimentation est un autre moyen d'alimenter la carte. On utilise une alimentation externe (de 7 à 12 V) branchée sur cette fiche, comme un adaptateur AC-DC relié à une prise électrique, ou une source portable, comme une batterie, une pile, ou un panneau solaire. En fait, ce connecteur est souvent utilisé lorsqu'on souhaite que notre circuit fonctionne de manière autonome sans être connecté à un ordinateur.



. Les LED

La carte Arduino Uno possède quatre LED qui servent à donner des indications sur son fonctionnement :

- “**ON**” s’allume lorsque la carte Arduino est alimentée,



- “**RX**” et “**TX**” s’allument, respectivement, lorsque la carte reçoit et transmet des données par le port série (COM),

- “**L**” est connectée à la broche 13. Elle s’allume lorsque l’état logique de cette broche est à un niveau haut. Ce connecteur est souvent utilisé pour réaliser des tests.

. Le bouton reset

Le bouton reset est un petit interrupteur qui vous permet de réinitialiser la carte ou de l’arrêter lorsqu’il est maintenu enfoncé quelques secondes. On peut, par ailleurs, obtenir le même résultat en connectant la broche “reset”, située à côté de la broche “3.3V”, à la masse (une des broches “GND”).

1.3. Le logiciel "IDE ARDUINO" - Initiation au langage ARDUINO

L'Arduino est une carte électronique qui ne sait rien faire sans qu'on lui dise quoi faire. Pourquoi ? Eh bien c'est dû au fait qu'elle est programmable. Cela signifie qu'elle a besoin d'un programme pour fonctionner.

Voici un exemple de programme :

```
// définition de la broche 2 de la carte en tant que variable
const int led_rouge = 2;

// fonction d'initialisation de la carte
void setup()
{
    // initialisation de la broche 2 comme étant une sortie
    pinMode(led_rouge, OUTPUT);
}

void loop()
{
    // allume la LED
    digitalWrite(led_rouge, LOW);
    // fait une pause de 1 seconde
    delay(1000);
    // éteint la LED
    digitalWrite(led_rouge, HIGH);
    // fait une pause de 1 seconde
    delay(1000);
}
```

Ce programme a été édité avec le logiciel "**IDE ARDUINO**" qui est disponible à l'adresse suivante :

https://www.arduino.cc/en/Main/Software

Le logiciel "**IDE ARDUINO**" est un logiciel de programmation pour l'Arduino, dans un langage dérivé du C /C++.

IDE veut dire : Integrated Development Environment ou en français, environnement de développement intégré.

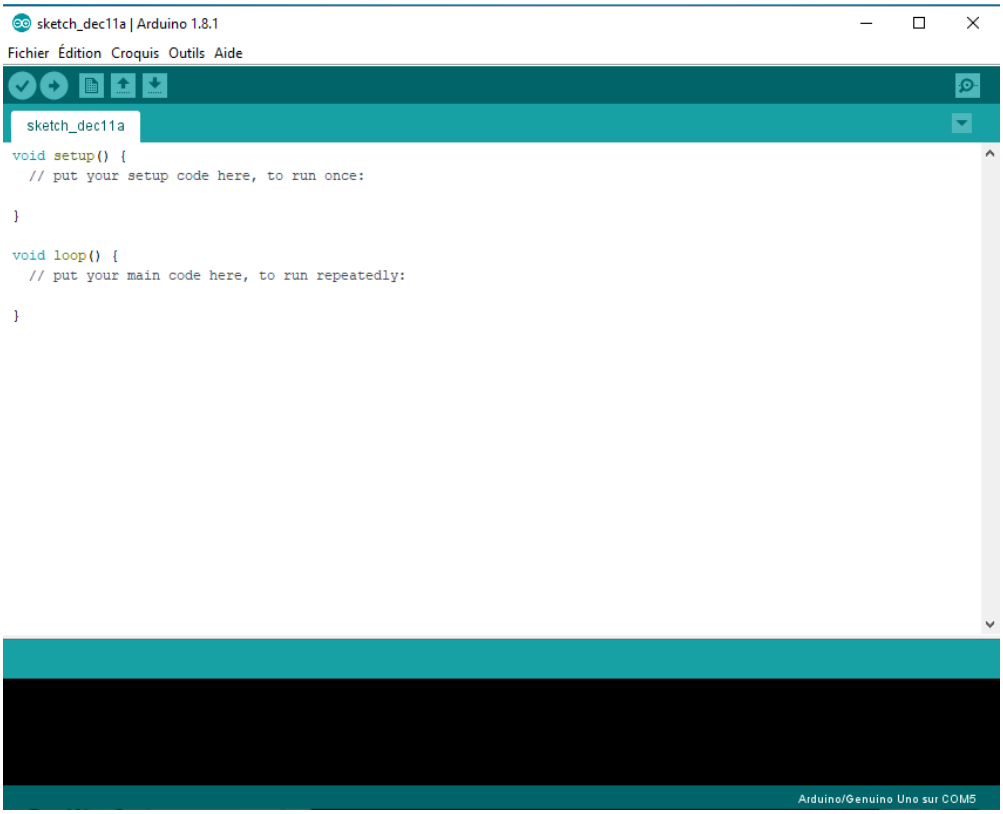
Et bien-sûr, pour programmer une carte Arduino, celle-ci doit être au préalable connectée à un ordinateur à l'aide d'un câble USB type A-B.

L'IDE Arduino est une interface graphique simple d'utilisation qui regroupe tous les outils qui permettent de programmer l'Arduino.

. Installation et exécution du logiciel "IDE ARDUINO"

Une fois téléchargé, double-cliquez sur l'exécutable du fichier et suivez tout simplement le guide d'installation.

Quand le logiciel est installé, ouvrez-le, une fenêtre devrait s'afficher, confirmant ainsi que l'environnement de développement a été correctement installé.



Tout en haut de la fenêtre, vous avez la barre de menus qui regroupe toutes les actions que vous pouvez réaliser avec le logiciel.

Ce qui nous intéresse cependant est la zone des 5 icônes juste en dessous : ce seront les boutons que vous utiliserez le plus souvent lorsque vous programmez. Nous avons, de la gauche vers la droite :

- le bouton "**Vérifier**", pour vérifier votre programme : il faut en effet que le programme que vous avez écrit ne présente pas de bugs afin de s'exécuter correctement, 

- le bouton "**Téléverser**" : en cliquant sur ce bouton, vous transférez votre programme compilé dans la mémoire de votre carte Arduino, 

- le bouton “**Nouveau**” : c’est à l’aide de ce bouton que vous créez de nouveaux programmes, 

- le bouton “**Ouvrir**” qui vous permet d’accéder aux programmes d’exemple de l’IDE ou aux programmes présents sur votre ordinateur, 

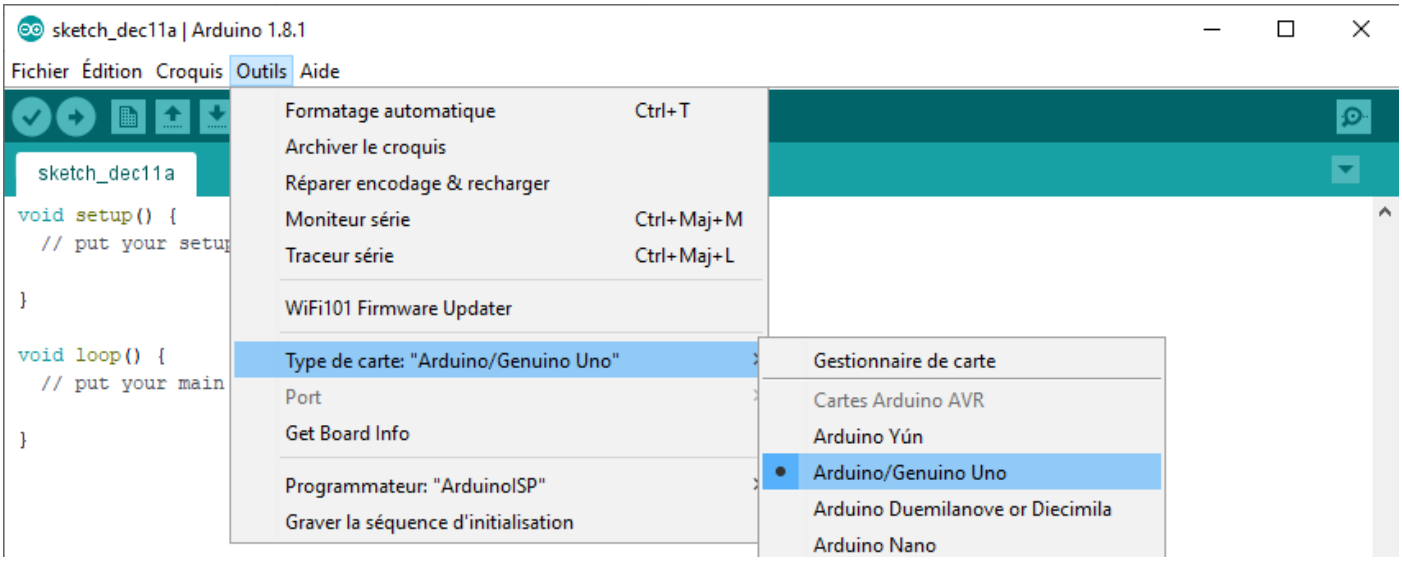
- le bouton “**Enregistrer**” par lequel vous pouvez sauvegarder le travail que vous avez réalisé afin d’y revenir quand vous le souhaitez. 

Tout en bas de la fenêtre du logiciel, se trouve la console de débogage : c’est là que vous trouverez les différentes erreurs d’exécution de votre programme le cas échéant. Grâce à cette console, vous vous assurez que votre programme fonctionne comme vous le souhaitez avant de le transférer sur votre carte Arduino.

<u>Attention</u> :

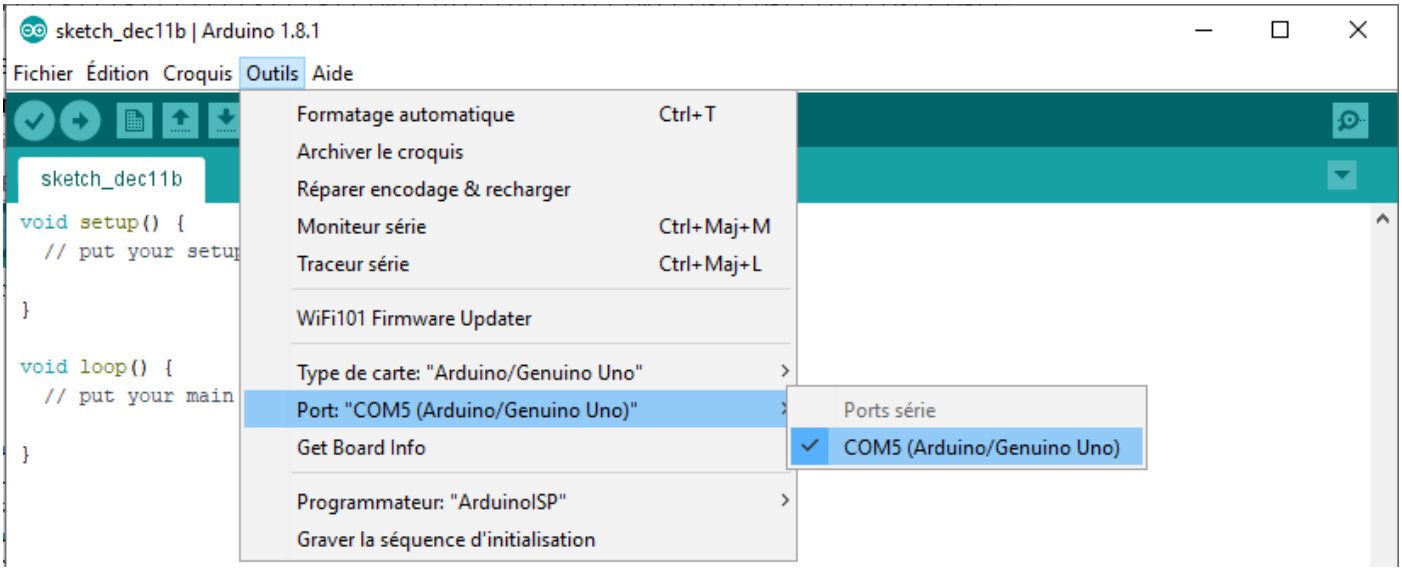
Avant de commencer à programmer, il faut d’abord indiquer au logiciel le type de carte qui recevra le programme.

Pour cela, cliquez sur le menu **”Outils”**, puis **“Type de carte”** et sélectionnez le type correspondant à la carte que vous avez branchée sur votre ordinateur :



Si votre carte n’est pas présente dans la liste, vous pouvez faire une recherche en cliquant plutôt sur **“Gestionnaire de carte”**. La fenêtre qui apparaît vous permet de chercher puis de télécharger le type de votre carte.

Une fois le type sélectionné, vous devez aussi indiquer le port sur lequel votre carte est branchée. Vous le trouverez dans le Menu **“Outils”** puis **“Port”** :



Vous pouvez maintenant commencer à programmer et envoyer votre programme sur l'Arduino en cliquant sur le bouton **"Téléverser"**. Après l'étape de compilation, le programme est transféré sur la carte. Vous aurez, au niveau de la console de débogage, un message vous indiquant que le téléversement est terminé.

Une fois le programme téléverser, il est conservé dans la mémoire de l'Arduino. Et dès que votre carte Arduino est alimentée, celle-ci exécute le programme qui est stocké dans sa mémoire.

Donc, lorsque vous branchez l'Arduino à l'ordinateur, le dernier programme reçu est exécuté. Et avant de commencer un nouveau montage, il est plus que conseillé de téléverser un programme d'initialisation, du modèle de celui-ci :

InitArduino

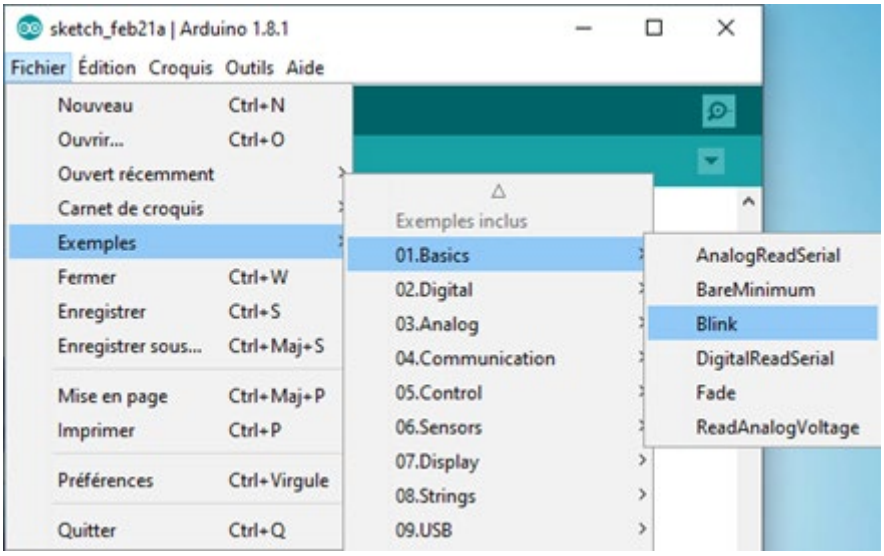
```
void setup() {

}

void loop() {

}
```

. Initiation au langage ARDUINO

Dans le logiciel **"ARDUINO IDE"**, de nombreux exemples pré-enregistrés peuvent être chargés et testés. Ils permettent de se familiariser plus rapidement avec le langage "ARDUINO".

Pour commencer, nous allons étudier le programme le plus simple. Il s'agit du programme « Blink » qui fait clignoter la Led L intégrée à la carte Arduino et connectée à la broche 13.

Pour ouvrir le programme : Faire Fichier > Exemples >01.Basics > Blink.



Le programme est alors affiché :

LED_BUILTIN correspond à la led qui se trouve sur la carte

La broche de la Led est initialisée en sortie numérique

```
void setup() {
// initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}


void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

Envoie 5V sur la LED de la carte

Attendre 1000 ms

Remet la tension de la led à 0V

. Structure de base d'un programme

On remarque que ce programme est séparé en 2 parties :

. une fonction **setup()**

Cette fonction est appelée au démarrage du programme. Elle est utilisée pour initialiser des variables préalablement déclarée, configurer le sens des broches.

La fonction setup n'est exécutée qu'une seule fois, après chaque mise sous tension ou reset (réinitialisation) de la carte Arduino.

Syntaxe de la fonction :

```
void setup()
{
}
```

Ici, on définit que la broche 13, reliée à la DEL "LED_BUILTIN", sera utilisée en sortie (OUTPUT).

. une fonction **loop()**

La fonction loop () (boucle en anglais) fait exactement ce que son nom suggère et exécute en boucle sans fin ses instructions pour contrôler la carte Arduino.

Syntaxe de la fonction :

```
void loop()
{
}
```

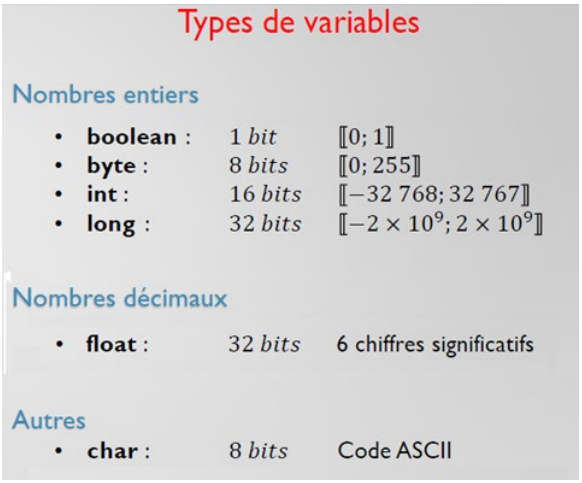
Ici elle applique une tension de 5V (HIGH) sur la broche 13, attend 1s (1000 ms), applique ensuite une tension de 0V (LOW) et attend encore 1s.

Remarques:

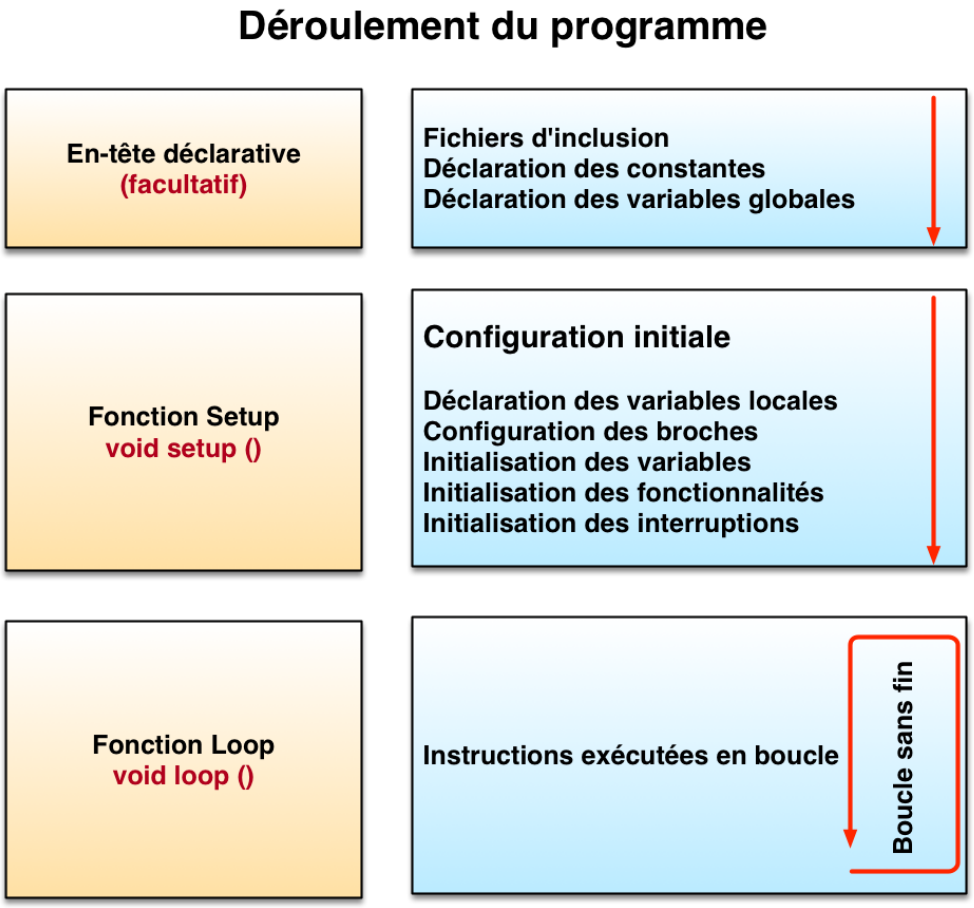
. Dans tout programme pour Arduino, codé en langage « Arduino », ces deux fonctions sont obligatoires.

. En C/C++, les instructions sont séparées par des points virgules « **;** » et les blocs de code sont encadrés par des accolades **{ }**.

. Une variable doit toujours être déclarée avec son type. Les types de variables couramment utilisés sont:



. Généralement, en plus des deux fonctions obligatoires, le programme comporte également au début, une partie déclarative (Inclusion des librairies, déclaration des constantes et variables globales) :



Les variables, déclarées en début de programme et en dehors des fonctions sont dites globales parce qu'elles sont accessibles et modifiables depuis n'importe quel endroit du code.

Après avoir vu le principe du déroulement d'un programme en langage Arduino, nous allons maintenant nous intéresser aux bases de la programmation.

1.4. Les bases de la programmation

Les programmes en langage Arduino, basé sur les langages C/C++, peuvent être divisés en trois parties principales :

. La structure,

. Les valeurs (variables et constantes),

. Les fonctions.

1.4.1. Structure du programme

Dans les fonctions de bases **”setup()”** et **”loop()”** obligatoires dans tous les programmes ou dans toute autre fonction, on utilisera les éléments de langage suivant :

1.4.1.1. Syntaxe de base

. Le point-virgule ;

Il est obligatoire à la fin de chaque instruction. Pour le compilateur, les sauts de lignes n'ont pas de signification : c'est le point-virgule qui marque la fin de ligne.
Oublier le point-virgule en fin de ligne donnera une erreur de compilation.

Exemple :

```
int a = 13;     // le point-virgule indique la fin de l'instruction
```

. Les accolades {}

Les accolades sont un élément majeur de la programmation en langage C.

Toute ouverture d'une accolade d'ouverture "{" doit obligatoirement être accompagnée dans le code d'une accolade de fermeture "}" correspondante. On dit souvent qu'il faut que les accolades soient équilibrées (càd autant de"{" que de "}").

L’Arduino IDE inclut une fonctionnalité pratique pour vérifier la correspondance des accolades entre elles. Il suffit de sélectionner une accolade, ou même de cliquer juste après une accolade, et l’accolade d’ouverture ou de fermeture associée sera mise en surbrillance.

Les accolades sont utilisées dans les fonctions, les boucles et les conditions

Exemples :

- Avec des fonctions :

```
void myfunction(datatype argument){        // ouverture de la fonction
        // instructions
}      // fermeture de la fonction
```

- Dans des boucles :

```
while (boolean expression)
{  // accolade d'ouverture du code de la boucle while
            // instructions
}  // fermeture de la boucle

do
 {    // accolade d'ouverture du code de la boucle do
             // instructions
 } while (boolean expression);  // fermeture de la boucle

for (initialisation; termination condition; incrementing expr)
{  // ouverture du code de la boucle for
            //instructions
  }  // fermeture de la boucle
```

- Dans des conditions :

```
if (boolean expression)
{ // ouverture du code de la condition if
             // instructions
} // fermeture du code de la condition if

else if (boolean expression)
{ // ouverture du code de l'instruction else if
             // instructions
} // fermeture du code de l'instruction else if

        else
           { // ouverture du code de l'instruction else
             // instructions
           } // fermeture du code de l'instruction else
```

. // Les commentaires

Lorsqu'on écrit dans le programme derrière un double slash **//**, c'est pour donner des informations sur le programme. Les commentaires n'ont aucune action sur le programme.

On peut écrire un commentaire sur plusieurs lignes entre **/* et */** .

Exemple : /*commentaire sur plusieurs lignes, commentaire sur plusieurs lignes,
commentaire sur plusieurs lignes */

1.4.1.2. Les opérateurs arithmétiques

. opérateur d'assignement (signe égal unique) =

Cet opérateur stocke la valeur à droite du signe égal dans la variable à gauche du signe égal.

Le signe égal dans le langage de programmation C est appelé opérateur d'assignement. Il a un sens différent de celui qu'il a en algèbre où il indique une équivalence ou une égalité. L'opérateur d'assignement indique au microcontrôleur d'évaluer la valeur ou l'expression qui se trouve à droite du signe égal et de la stocker dans la variable à gauche du signe égal :

variable = valeur ;

Exemple :

```
int sensVal;              // déclare une variable entière de 16 bits nommée senVal
senVal = analogRead(0);   // mémorise la valeur de la conversion analogique numérique
                                dans la variable
```

. Addition + , Soustraction - , Multiplication * , et Division /

Ces opérateurs renvoient respectivement la somme, la différence, le produit ou le quotient entre deux opérandes (= entre deux termes).

Cette opération est réalisée en utilisant le type des données des opérandes. Ainsi par exemple, 9 / 4 donne 2 dès lors que 9 et 4 sont de type int.

Si les opérandes sont de deux types de données différents, le plus "grand" est utilisé pour le calcul.

Si un des nombres (opérandes) est du type float ou double, le calcul dit "en virgule flottante" est utilisé pour le calcul (càd que 9/4 donne 2.25).

- Syntaxe :

```
result = value1 + value2;
result = value1 - value2;
result = value1 * value2;
result = value1 / value2;
```

- Exemple :

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

1.4.1.3. Les opérateurs de comparaison

==, !=, <, > sont des opérateurs logiques de comparaison :

- x == y (x est égal à y)
- x != y (x est différent de y)
- x < y (x est inférieur à y)
- x > y (x est supérieur à y)
- x <= y (x est inférieur ou égal à y)
- x >= y (x est supérieur ou égal à y)

1.4.1.4. Les opérateurs booléens

Ces opérateurs peuvent être utilisés à l'intérieur de la condition d'une instruction if pour associer plusieurs conditions (ou opérandes) à tester.

. && (ET LOGIQUE)

VRAI seulement si les deux opérandes sont VRAI, par exemple :

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) { // lit l'état de 2 entrées
  // ...
}
// est VRAI seulement si les deux entrées sont à l'état HAUT simultanément.
```

. || (OU LOGIQUE)

VRAI si l'un des deux opérandes est VRAI, par exemple :

```
if (x > 0 || y > 0) { // si x supérieur à 0 ou si y supérieur à 0
  // ...
}
 // est VRAI si soit x, soit y est supérieur à 0.
```

. ! (NON LOGIQUE)

VRAI si l'opérande est FAUX, par exemple :

```
if (!x) {
  // ...
}
  // VRAI si la variable x est FAUSSE (càd si x = 0)
```

1.4.1.5. Les opérateurs composés

. ++ (incrément) / -- (décrément)

Ces opérateurs incrémentent ou décrémentent une variable entière de type int ou long (pouvant être unsigned).

- Syntaxe :

```
x++;  // incrémente x de un, sans modifier x
++x;  // incrémente x de un et modifie x

x-- ;  // décrément x de un, sans modifier x
--x ;  // décrémente x de un, et modifie x
```

- Exemples :

```
x = 2; // variable x = 2
y = ++x;     // x contient 3 et y contient 3
y = x++;     // x contient toujours 2, y contient 3
```

. += , -= , *= , /=

Ces opérateurs réalisent une opération mathématique entre une variable **x** et une autre variable ou une constante **y**. Les opérateurs **+,-** ,***** et **/** sont juste utiles pour raccourcir la forme complète des opérations mathématiques listées ci-dessous :

```
x += y;   // équivaut à l'expression x = x+y
x -= y;   // équivaut à l'expression x = x- y
x *= y;   // équivaut à l'expression x = x * y
x /= y;   // équivaut à l'expression x= x / y
```

- Exemples :

```
x = 2;
x += 4;       // x contient 6
x -= 3;       // x contient 3
```

```
x *= 10;      // x contient 30
x /= 2;       // x contient 15
```

1.4.1.6. Structures de contrôle

. if (condition)

L'instruction if ("si" en français), utilisée avec un opérateur logique de comparaison, permet de tester si une condition est vraie, par exemple si la mesure d'une entrée analogique est bien supérieure à une certaine valeur.

Le format d'un test if est le suivant :

```
if (uneVariable > 50)
{
  // faire quelque chose
}
```

Dans cet exemple, le programme va tester si la variable **uneVariable** est supérieure à 50. Si c'est le cas, le programme va réaliser une action particulière. Autrement dit, si l'état du test entre les parenthèses est vrai, les instructions comprises entre les accolades sont exécutées. Sinon, le programme se poursuit sans exécuter ces instructions.

Les accolades peuvent être omises après une instruction if. Dans ce cas, la suite de la ligne (qui se termine par un point-virgule) devient la seule instruction de la condition. Tous les exemples suivants sont corrects :

```
if (x > 120)  digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }
```

. if / else

L'instruction if/else (si/sinon en français) permet un meilleur contrôle du déroulement du programme que la simple instruction if, en permettant de grouper plusieurs tests ensemble. Par exemple, une entrée analogique peut-être testée et une action réalisée si l'entrée est inférieure à 500, et une autre action réalisée si l'entrée est supérieure ou égale à 500. Le code ressemblera à cela :

```
if (valsensor < 500)
{
  // action A
}
else
{
  // action B
}
```

L'instruction **else** peut contenir un autre test **if**, et donc des tests multiples, mutuellement exclusifs peuvent être réalisés en même temps.

Chaque test sera réalisé après le suivant jusqu'à ce qu'un test VRAI soit rencontré.
Quand une condition vraie est rencontrée, les instructions associées sont réalisées, puis le programme continue son exécution à la ligne suivant l'ensemble de la construction **if/else**.
Si aucun test n'est VRAI, le bloc d'instructions par défaut **else** est exécuté, s'il est présent, déterminant ainsi le comportement par défaut.

Un bloc **else if** peut être utilisé avec ou sans bloc de conclusion **else**.
Un nombre illimité de branches **else if** est autorisé.

```
if (valsensor < 500)
{
  // action A
}
else if (valsensor >= 1000)
{
  // action B
}
else
{
  // action C
}
```

. Boucle for

L'instruction **for** est utilisée pour répéter l'exécution d'un bloc d'instructions regroupées entre des accolades.
Un compteur incrémental est habituellement utilisé pour incrémenter et finir la boucle.
L'instruction **for** est très utile pour toutes les opérations répétitives et est souvent utilisées en association avec des tableaux de variables pour agir sur un ensemble de données ou broches.

Il y a 3 parties dans l'entête d'une boucle for :

```
for (initialisation; condition; incrementation) {

        //instructions

}
```

L'initialisation a lieu en premier et une seule fois. A chaque exécution de la boucle, la condition est testée.
Si elle est VRAIE, le bloc d'instructions et l'incrémentation sont exécutés, puis la condition est testée de nouveau. Lorsque la condition devient FAUSSE, la boucle stoppe.

Exemple :

```
for (int i=0; i <= 255; i++){   // boucle incrémentant la variable i de 0 à 255, de 1 en 1

        //instructions

  } // fin de la boucle for
```

. Boucle while

Les boucles **while** ("tant que" en anglais) bouclent sans fin, et indéfiniment, jusqu'à ce que la condition ou l'expression entre les parenthèses ( ) devienne fausse.
Quelque chose doit modifier la variable testée, sinon la boucle **while** ne se terminera jamais. Cela peut être dans votre code, soit une variable incrémentée, ou également une condition externe comme le test d'un capteur.

```
while(expression){   // tant que l'expression est vraie

          // instructions à effectuer

}
```

Avec **expression,** une instruction (booléenne) qui renvoie un résultat VRAI ou FAUX

Exemple :

```
var = 0;
while(var < 200){  // tant que la variable est inférieur à 200

       // fait quelque chose 200 fois de suite...

  var++; // incrémente la variable

}
```

. boucle do – while

La boucle **do / while** ("faire tant que" en anglais) fonctionne de la même façon que la boucle **while**, à la différence près que la condition est testée à la fin de la boucle, et par conséquent la boucle do sera toujours exécutée au moins une fois.

- Syntaxe :

```
do // faire...
{
   // instructions

} while (condition); // tant que la condition est vraie
```

Avec **condition**, une expression booléenne dont le résultat peut être VRAI ou FAUX.

- Exemple :

```
do // faire...
{
   x = analogRead(A0);  // lit la valeur de la tension d'un capteur

} while (x < 100); // ...tant que x est inférieur à 100
```

Remarque :

L'instruction **break** est utilisée pour sortir d'une boucle do, for ou while, en passant outre le déroulement normal de la boucle.

Exemple :

```
for (int i=0; i <= 100; i++){   // boucle incrémentant la variable i de 0 à 100, de 1 en 1

 x = analogRead(A0);  // lit la valeur de la tension d'un capteur
     if (x < 100)
    {
    break ; // si la mesure est inférieure à un seuil, on sort de la boucle
    }
 } // fin de la boucle for
```

1.4.2. Variables et constantes

Les variables sont des expressions que l'on utilise dans les programmes pour stocker des valeurs, telles que la tension de sortie d'un capteur présente sur une broche analogique.

1.4.2.1. Les constantes Arduino prédéfinies

Dans le langage Arduino, les constantes sont des variables prédéfinies. Elles sont utilisées pour rendre les programmes plus faciles à lire.

. INPUT ET OUTPUT

Ces constantes sont utilisées pour définir des broches numériques en entrée ou en sortie :

Les broches numériques peuvent être utilisées soit en mode **INPUT** (= en entrée), soit en mode **OUTPUT** (= en sortie).

Modifier le mode de fonctionnement d'une broche du mode **INPUT** (=ENTREE) en mode **OUTPUT**(=SORTIE) avec l'instruction **pinMode()** change complètement le comportement électrique de la broche.

- *Broches configurées en entrée (INPUT)*

Les broches d'une carte Arduino configurées en mode **INPUT** (=en entrée) à l'aide de l'instruction **pinMode()** sont dites en état de "haute-impédance".

Ces broches ne consomment alors qu'une toute petite intensité (de l'ordre du microampère) du circuit sur lequel elles sont connectées. Leur mode de fonctionnement est équivalent à celui d'un voltmètre.

- *Broches configurées en sortie (OUTPUT)*

Les broches configurées en mode **OUTPUT** (= en sortie) avec l'instruction **pinMode()** sont en état dit de "basse-impédance" .

Cela veut dire qu'elles peuvent fournir une quantité significative de courant aux autres circuits. Chaque broche de la carte Arduino peut fournir jusqu'à 40 mA d'intensité au circuit sur lequel elle est connectée. Son mode de fonctionnement est équivalent à celui d'un générateur.

Cependant, l'intensité cumulée fournie par les broches de la carte ne doit pas dépasser les 200mA !

. HIGH ET LOW

Lorsqu'on lit ou on écrit sur une broche numérique, seuls deux états distincts sont possibles, la broche ne peut être qu'à deux valeurs : **HIGH (HAUT) ou LOW (BAS).**

HIGH

La signification de la constante **HIGH** (en référence à une broche) est quelque chose de différent selon que la broche est définie comme une ENTREE ou comme une SORTIE.

Si la broche est configurée en ENTREE avec l'instruction pinMode, et lue avec l'instruction digitalRead, le microcontrôleur renverra HIGH (=HAUT) si une tension de 3V ou + est présente sur la broche.

Quand une broche est configurée en SORTIE avec l'instruction pinMode, et mise au niveau HAUT avec l'instruction digitalWrite, la broche est mise à 5V. Dans cet état, la broche peut fournir une certaine intensité (jusqu'à 40 mA par broche, sans dépasser 200 mA pour l'ensemble des broches).

LOW

La constante LOW a également une signification différente selon que la broche est configurée en ENTREE ou en SORTIE.

Quand une broche est configurée en ENTREE avec l'instruction pinMode, et lue avec l'instruction digitalRead, le microcontrôleur renverra un niveau BAS si une tension de 2V ou moins est présente sur la broche.

Quand la broche est configurée en SORTIE avec l'instruction pinMode, et est mise au niveau LOW avec l'instruction digitalWrite, la broche est à 0 volts.

. true ET false

Il existe deux constantes utilisées pour représenter le VRAI et le FAUX dans le langage Arduino : true et false.

false *(= FAUX)*

La constante false est définie comme le 0 (zéro).

true *(=VRAI)*

La constante true est définie comme tout entier qui n'est pas 0 (zéro).

Noter que les constantes **true** et **false** sont écrites en minuscule à la différence des constantes **HIGH**, **LOW**, **INPUT** et **OUTPUT**.

1.4.2.2. Les variables - Types de données

Les variables peuvent être de type variés. Les différents types de variables sont décrits ci-dessous :

. Int

Déclare une variable de type int (pour integer, entier en anglais). Les variables de type int sont le type de base pour le stockage de nombres, et ces variables stockent une valeur sur 2 octets. Elles peuvent donc stocker des valeurs allant de - 32 768 à 32 767 (valeur minimale de -2^{15} et une valeur maximale de $2^{15}-1$).

- Syntaxe :

```
int var = val;
```

- **var** : le nom de votre variable de type int
- **val** : la valeur d'initialisation de la variable

- Exemple :

```
int ledPin = 13; // déclare une variable de type int appelée LedPin et valant 13
```

- Remarque :

Quand les variables dépassent la valeur maximale de leur capacité, elles "débordent" et reviennent à leur valeur minimale, et ceci fonctionne dans les 2 sens :

```
int x   // déclaration de la variable de type int appelée x
x = -32,768;  // x prend la valeur -32 768
x = x - 1;      // x vaut maintenant 32 767, car déborde dans le sens négatif

x = 32,767;  // x prend la valeur 32 767
x = x + 1;      // x vaut maintenant la valeur - 32 768, car déborde dans le sens positif
```

. unsigned int

Déclare une variable de type int non-signée. Les variables de type unsigned int (entiers non signée) sont les mêmes que les variables de type int en ce sens qu'elle stocke une valeur sur 2 octets. Cependant, au lieu de stocker des valeurs négatives, les variables de

type unsigned int stocke uniquement des valeurs positives, dans une fourchette allant de 0 à 65535 (2^{16}-1).

- Syntaxe :

```
unsigned int var = val;
```

- **var** : le nom de votre variable int
- **val** : la valeur donnée à votre variable

- Exemple :

```
unsigned int ledPin = 13; // déclaration d'une variable entière non signée nommée ledPin et valant 13
```

- Remarque :

Quand la valeur des variables excède leur capacité maximale, elles "débordent" et reprennent leur valeur minimale, et ceci se produit dans les 2 sens.

```
unsigned int x  // déclaration d'une variable int non signée nommée x
  x = 0;            // x vaut 0
  x = x - 1;      // x contient maintenant 65535
  x = x + 1;      // x contient maintenant 0
```

. long

Déclare des variables de type long. Les variables de type long sont des variables de taille élargie pour le stockage de nombre entiers, sur 4 octets (32 bits), de -2 147 483 648 à + 2 147 483 647.

- Syntaxe :

```
long var = valeur;
```

. var : le nom de la variable long

. valeur : la valeur donnée à la variable

- Exemple :

```
long speedOfLight = 186000L;   //déclare une variable de type long
   // le 'L' pour forcer à traiter la variable dans le format de donnée de type long
```

. unsigned long

Déclare une variable de type long non signé. Les variables de type long non signé sont des variables de taille élargie pour le stockage de nombre entier qui stocke les valeurs sur 4 octets (32 bits).

A la différence des variables de type long standard, les variables de type long unsigned ne peuvent pas stocker des nombres négatifs, la fourchette des valeurs qu'elles peuvent stocker s'étendant de 0 à 4 294 967 295 (2^{32}-1)

- Syntaxe :

```
unsigned long var = valeur;
```

. var : le nom de votre variable de type long

. valeur : la valeur donnée à la variable

- Exemple :

```
unsigned long time; // déclare une variable de type long non signé appelée time
void setup()
{
  Serial.begin(9600); // initialise la connexion série à 9600 bauds
}
void loop()
{
  Serial.print("Time: ");
  // Met dans la variable time le temps écoulé depuis le démarrage
  time = millis();
 //affiche le temps écoulé depuis que le programme a démarré
  Serial.println(time);
```

```
// attend une seconde avant d'envoyer un nouveau message au PC
  delay(1000);
}
```

. float

Déclare des variables de type "virgule-flottante", c'est à dire des nombres à virgules. Les nombres à virgule sont souvent utilisés pour l'expression des valeurs analogiques.

Les nombres à virgule ainsi stockés peuvent prendre des valeurs entre - 3.4028235E+38 et 3.4028235E+38. Ils sont stockés sur 4 octets (32 bits) de mémoire.

Les variables float ont seulement 6 à 7 chiffres de précision. Ceci concerne le nombre total de chiffres, pas seulement le nombre à droite de la virgule.

- Syntaxe :

```
float var = valeur;
```

. var : le nom de la variable

. valeur : la valeur donnée à la variable

- Exemple :

```
float myfloat; // déclare une variable à virgule appelée myfloat
float sensorCalibrate = 1.117; // déclare une variable à virgule appelée sensorCalibrate

int x; // déclare une variable entière de type int appelée x
int y; // déclare une variable entière de type int appelée y
float z; // déclare une variable nombre à virgule de type float appelée z

x = 1;            // x vaut 1
y = x / 2;        // y vaut 0 car les entiers ne supporte pas les décimales
z = float (x)/ 2.0;   // z vaut 0.5 (float (x) : conversion de x en float, alors x=1.0)
```

. char

Déclare une variable d'un octet de mémoire (8 bits) qui contient une valeur correspondant à un caractère. Les caractères unitaires sont écrits entre guillemets uniques, comme ceci : **'A'**.

Les caractères sont stockés de la même façon que les nombres. A chaque caractère correspond une valeur numérique comprise entre 0 et 127 (code ASCII) :

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

La variable de type char est de type signée, ce qui veut dire qu'elle peut contenir des valeurs allant de -128 à +127.

- Syntaxe :

```
char monChar='B'; // déclare une variable char
```

- Exemple :

```
char myChar = 'A'; // déclare une variable char initialisée avec la valeur A
char myChar = 65;     // expression équivalente car la valeur ASCII de A est 65
```

1.4.3. Les fonctions

Une fonction est un bloc d'instruction que l'on peut appeler à tout endroit d'un programme.

En langage Arduino, des bibliothèques de fonctions prédéfinies sont disponibles afin de manipuler facilement les entrées/sorties, gérer le temps, gérer la communication série…

On peut également créer ses propres fonctions qui seront utiles pour l'exécution de tâches répétitives et évitant alors la réécriture des lignes de codes à chaque fois que se présente ces tâches.

1.4.3.1. Fonctions prédéfinies des Entrées/Sorties numériques

. pinMode()

Configure la broche spécifiée pour qu'elle se comporte soit en entrée, soit en sortie.

- Syntaxe :

pinMode(broche, mode)

- Paramètres :

broche: le numéro de la broche de la carte Arduino dont le mode de fonctionnement (entrée ou sortie) doit être défini.

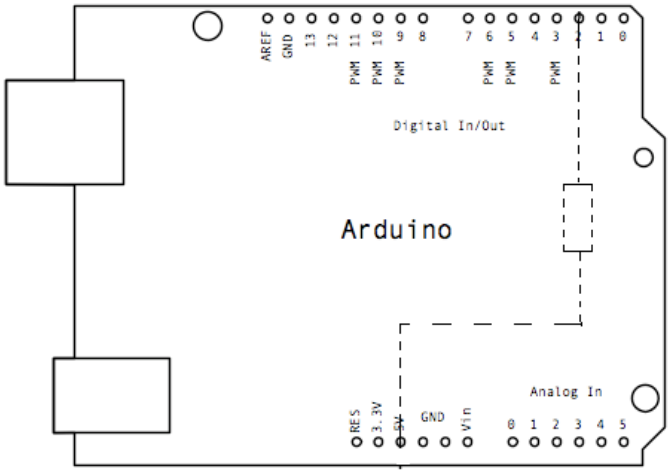
mode: soit INPUT (entrée en anglais) ou OUTPUT (sortie en anglais)

. digitalWrite()

Met un niveau logique HIGH (HAUT en anglais) ou LOW (BAS en anglais) sur une broche numérique.

Si la broche a été configurée en SORTIE avec l'instruction pinMode(), sa tension est mise à la valeur correspondante : 5V pour le niveau HAUT, 0V (masse) pour le niveau BAS.

Si la broche est configurée en ENTREE, écrire un niveau HAUT sur cette broche a pour effet d'activer la résistance interne de 20K sur cette broche.



A l'inverse, mettre un niveau BAS sur cette broche configurée en ENTREE désactivera la résistance interne.

- Syntaxe :

digitalWrite(broche, valeur)

- Paramètres :

broche: le numéro de la broche de la carte Arduino

valeur : HIGH ou LOW (ou bien 1 ou 0)

- Exemple :

```
int ledPin = 13;              // LED connectée à la broche numérique n° 13

void setup()
{
  pinMode(ledPin, OUTPUT);     // met la broche utilisée avec la LED en SORTIE
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // allume la LED
  delay(1000);                  // pause 1 seconde
  digitalWrite(ledPin, LOW);    // éteint la LED
  delay(1000);                  // pause 1 seconde
}
```

Remarques :

- Les broches analogiques peuvent être utilisées en tant que broches numériques, représentées par les nombres 14 (entrée analogique 0) à 19 (entrée analogique 5).

- Cette instruction met la valeur 0/1 dans le bit de donnée qui est associé à chaque broche, ce qui explique qu'on puisse le mettre à 1, même si la broche est en entrée.

- Ne pas oublier qu'une broche numérique ne peut fournir que 40mA (milliampères) tant en entrée qu'en sortie, et que l'ensemble des broches de la carte Arduino ne peut fournir que 200mA. Par conséquent, limiter l'intensité utilisée pour chaque broche à une dizaine de mA par des résistances adaptées : 220 Ohms pour une LED par exemple.

. digitalRead()

Lit l'état (= le niveau logique) d'une broche précise en entrée numérique, et renvoie la valeur HIGH (HAUT en anglais) ou LOW (BAS en anglais).

- Syntaxe :

digitalRead(broche)

- Paramètres :

broche : le numéro de la broche numérique que vous voulez lire. (int)

- Valeur retournée :

Renvoie la valeur HIGH (HAUT en anglais) ou LOW (BAS en anglais)

- Exemple :

```
int ledPin = 13; // LED connectée à la broche n°13
int inPin = 7;   // un bouton poussoir connecté à la broche 7
int val = 0;     // variable pour mémoriser la valeur lue

void setup()
{
  pinMode(ledPin, OUTPUT);      // configure la broche 13 en SORTIE
  pinMode(inPin, INPUT);      // configure la broche 7 en ENTREE
}
void loop()
{
  val = digitalRead(inPin);   // lit l'état de la broche en entrée et met le résultat dans la variable
  digitalWrite(ledPin, val);    // met la LED dans l'état du BP  (càd allumée si appuyé et inversement)
}
```

Dans ce programme, la broche 13 reflète fidèlement l'état de la broche 7 qui est une entrée numérique.

Remarques :

- Si la broche numérique en entrée n'est connectée à rien, l'instruction digitalRead() peut retourner aussi bien la valeur HIGH (HAUT en anglais) ou LOW (BAS en anglais) (et cette valeur peut changer de façon aléatoire)

- Les broches analogiques peuvent être utilisées en entrée numériques et sont désignées par les numéros 14 (entrée analogique 0) à 19 (entrée analogique 5).

1.4.3.2. Fonctions prédéfinies des Entrées/Sorties analogiques

. analogRead()

Lit la valeur de la tension présente sur la broche spécifiée. La carte Arduino comporte 6 voies connectées à un convertisseur analogique-numérique 10 bits. Cela signifie qu'il est possible de transformer la tension d'entrée entre 0 et 5V en une valeur numérique entière comprise entre 0 et 1023. Il en résulte une résolution (écart entre 2 mesures) de : 5 volts / 1024 intervalles, autrement dit une précision de 0.0049 volts (4.9 mV) par intervalle.

Une conversion analogique-numérique dure environ 100 µs pour convertir l'entrée analogique, et donc la fréquence maximale de conversion est environ de 10 000 fois par seconde.

- Syntaxe :

  analogRead(broche_analogique)

- Paramètres :

  broche_analogique : le numéro de la broche analogique (et non le numéro de la broche numérique) sur laquelle il faut convertir la tension analogique appliquée (0 à 5 sur la plupart des cartes Arduino)

- Valeur retournée :

  valeur int (0 to 1023) correspondant au résultat de la mesure effectuée

- Exemple :

```
int analogPin = 3;              // Capteur analogique relié à la broche A3
int val = 0;                    // variable de type int pour stocker la valeur de la mesure

void setup()
{
  Serial.begin(9600);       //  initialisation de la connexion série
}

void loop()
{

  // lit la valeur de la tension analogique présente sur la broche
  val = analogRead(analogPin);

   // affiche la valeur (comprise en 0 et 1023) dans la fenêtre terminal PC
  Serial.println(val);

}
```

Remarques :

- La tension de référence par défaut est le 5V : il est possible d'utiliser une autre valeur si besoin.

- Les broches analogiques sont utilisées en entrée. Il n'est pas nécessaire de les configurer au préalable à l'appel de la fonction analogRead

- Si la broche analogique est laissée non connectée, la valeur renvoyée par la fonction **analogRead()** va fluctuer en fonction de plusieurs facteurs (tels que la valeur des autres entrées analogiques, la proximité de votre main vis à vis de la carte Arduino, etc.).

. analogWrite()

Génère un signal carré de fréquence 490 Hz sur une broche de la carte Arduino (onde PWM - Pulse Width Modulation en anglais ou MLI - Modulation de Largeur d'Impulsion en français).

Après avoir appelé l'instruction analogWrite(), la broche générera un signal carré stable avec un rapport cyclique (fraction de la période où la broche est au niveau haut) de durée spécifiée (en %), jusqu'à l'appel suivant de l'instruction analogWrite() (ou bien encore l'appel d'une instruction digitalRead() ou digitalWrite() sur la même broche).

- Syntaxe :

analogWrite(broche, valeur);

- Paramètres :

- **broche**: la broche utilisée pour "écrire" l'impulsion. Cette broche devra être une broche ayant la fonction PWM, Par exemple, sur la UNO, ce pourra être une des broches 3, 5 ,6 ,9 ,10 ou 11.

- **valeur**: la largeur du "duty cycle" (proportion de l'onde carrée qui est au niveau HAUT) : entre 0 (0% HAUT donc toujours au niveau BAS) et 255 (100% HAUT donc toujours au niveau HAUT).

- Exemple :

Le programme suivant fixe la luminosité d'une LED proportionnellement à la valeur de la tension lue depuis un potentiomètre :

```
int ledPin = 9;      // LED connectée sur la broche 9
int analogPin = 3;   // le potentiomètre connecté sur la broche analogique 3
int val = 0;         // variable pour stocker la valeur de la tension lue

void setup()
{
  pinMode(ledPin, OUTPUT);   // configure la broche en sortie
}

void loop()
{
  val = analogRead(analogPin);   // lit la tension présente sur la broche en entrée
  analogWrite(ledPin, val / 4);  // Résultat d'analogRead entre 0 to 1023,
                                 // résultat d'analogWrite entre 0 to 255
                                 // => division par 4
}
```

- **Remarques** :

- Il n'est pas nécessaire de faire appel à l'instruction pinMode() pour mettre la broche en sortie avant d'appeler la fonction analogWrite().

- L'impulsion PWM générée sur les broches 5 et 6 pourront avoir des "duty cycle" plus long que prévu. La raison en est l'interaction avec les instructions millis() et delay(), qui partagent le même timer interne que celui utilisé pour générer l'impulsion de sortie PWM.

1.4.3.3. Fonctions prédéfinies des Entrées/Sorties avancées

. tone()

Génère une onde carrée (onde symétrique avec rapport cyclique (niveau haut/période) à 50%), à la fréquence spécifiée en Hertz (Hz) sur une broche.

La durée peut être précisée, sinon l'impulsion continue jusqu'à l'appel de l'instruction noTone().

La broche peut être connectée à un buzzer piézoélectrique ou autre haut-parleur pour jouer des notes (les sons audibles s'étendent de 20Hz à 20 000Hz).

Une seule note peut être produite à la fois. Si une note est déjà jouée sur une autre broche, l'appel de la fonction tone() n'aura aucun effet (tant qu'une instruction noTone() n'aura pas eu lieu).

Si la note est jouée sur la même broche, l'appel de la fonction tone() modifiera la fréquence jouée sur cette broche.

- Syntaxe :

tone(broche, frequence)

tone(broche, frequence, durée)

- Paramètres :

broche : la broche sur laquelle la note est générée.

frequence : la fréquence de la note produite en hertz (Hz)

durée : la durée de la note en millisecondes (optionnel)

. noTone()

Stoppe la génération d'impulsion produite par l'instruction tone(). N'a aucun effet si aucune impulsion n'est générée.

- Syntaxe :

noTone(broche)

- Paramètres :

broche : la broche sur laquelle il faut stopper la note.

. pulseIn()

Lit la durée d'une impulsion (soit niveau HAUT, soit niveau BAS) appliquée sur une broche (configurée en ENTREE).

Par exemple, si le paramètre valeur est HAUT, l'instruction pulseIn() attend que la broche passe à HAUT, commence alors le chronométrage, attend que la broche repasse au niveau BAS et stoppe alors le chronométrage.

L'instruction renvoie la durée de l'impulsion en microsecondes. L'instruction s'arrête et renvoie 0 si aucune impulsion n'est survenue dans un temps spécifié.

Il est préférable de travailler avec des impulsions d'une durée de 10 microsecondes à 3 minutes.

- Syntaxe :

pulseIn(broche, valeur)

pulseIn(broche, valeur, delai_sortie)

- Paramètres :

broche : le numéro de la broche sur laquelle vous voulez lire la durée de l'impulsion.

valeur : le type d'impulsion à "lire" : soit HIGH (niveau HAUT) ou LOW (niveau BAS

delai_sortie (optionnel): le nombre de microsecondes à attendre pour début de l'impulsion. La valeur par défaut est 1 seconde. (type unsigned long)

- Valeur renvoyée :

La durée de l'impulsion (en microsecondes) ou 0 si aucune impulsion n'a démarrer avant le délai de sortie. (type unsigned long)

- Exemple :

```
int broche = 7; // variable de broche
unsigned long duree;  // variable utilisée pour stocker la durée

void setup()
{
  pinMode(broche, INPUT); // met la broche en entrée
```

```
}

void loop()
{
  duree = pulseIn(broche, HIGH); // met la durée de l'impulsion de niveau HAUT dans la variable duree
}
```

1.4.3.4. Fonctions prédéfinies de gestion du temps

. delay(ms)

Réalise une pause dans l'exécution du programme pour la durée (en millisecondes) indiquée en paramètre.

- Syntaxe :

delay (ms);

- Paramètres :

ms (unsigned long): le nombre de millisecondes que dure la pause

. unsigned long millis()

Renvoie le nombre de millisecondes depuis que la carte Arduino a commencé à exécuter le programme courant. Ce nombre débordera (càd sera remis à zéro) après 50 jours approximativement.

- Syntaxe :

variable_unsigned_long = millis();

- Exemple :

```
unsigned long time;

void setup(){
```

```
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();
  //affiche sur le PC le temps depuis que le programme a démarré
  Serial.println(time);
  // pause d'une seconde afin de ne pas envoyer trop de données au PC
  delay(1000);
}
```

1.4.3.5. Fonctions prédéfinies de communication

La librairie **Serial** est utilisée pour les communications par le port série entre la carte Arduino et un ordinateur ou d'autres composants.

Le port série communique sur les broches 0 (RX) et 1 (TX) avec l'ordinateur via le port USB. C'est pourquoi, si on utilise cette fonctionnalité, il n'est pas possible d'utiliser les broches 0 et 1 en tant qu'entrées ou sorties numériques.

Le logiciel Arduino IDE dispose d'un moniteur série, qui permet de recevoir et d'envoyer des informations via une liaison série. Il est particulièrement intéressant pour les tests des programmes puisqu'il permet d'afficher les valeurs des variables, les états logiques des entrées et des sorties de l'Arduino, etc…

Il suffit pour cela de cliquer sur le bouton du moniteur série dans la barre d'outils puis de sélectionner le même débit de communication que celui utilisé dans l'appel de la fonction begin() pour établir la liaison série.

Les principales fonctions de la librairie **Serial** :

. **begin();** (Configure la vitesse de transmission du port série)

. **print();** (Envoie une donnée sous forme de chaine de caractères sur le port série)

. **println();** (Envoie une donnée sur le port série et fait un saut à la ligne)

. **write();** (Ecrit des données binaires sur le port série. Ces données sont envoyées comme une série d'octets)

. **read();** (Lis les données contenues dans la mémoire tampon (buffer) du port série)

. **flush();** (Vide la mémoire tampon de la liaison série)

. **available();** (Donne le nombre d'octets (caractères) disponible pour lecture dans la file d'attente (buffer) du port série)

Une fois la liaison série établie dans la fonction **setup()** avec la fonction **Serial.begin()**, il est possible d'envoyer des données depuis la carte Arduino vers le moniteur série avec la fonction "**print()**" de la classe "**Serial**".

On peut envoyer différents types d'informations :

. Envoie d'une chaîne de caractères, sans saut de ligne à la fin :

Serial.print("Test");

. Envoie d'une chaîne de caractères, avec saut de ligne à la fin :

Serial.println("Test");

. Envoie de la valeur d'une variable, sans saut de ligne à la fin :

int a = 3;

Serial.print(a);

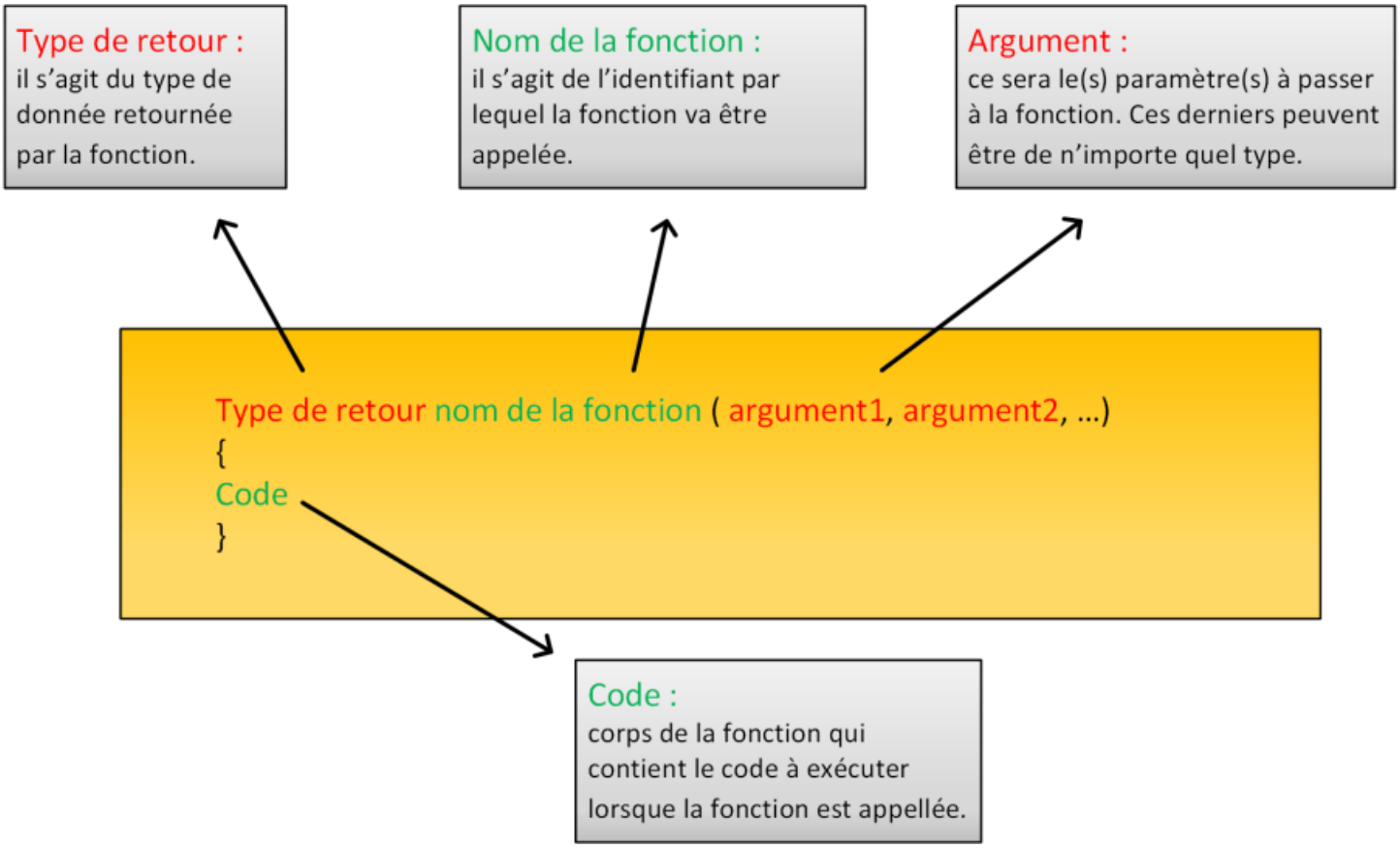
- Exemple :

```
int analogValue = 0; // variable pour stocker la valeur analogique sur la broche A0
void setup() {
// initialise le port série à 9600 bauds
Serial.begin(9600);
}
void loop() {
// lit la valeur analogique sur la broche A0
analogValue = analogRead(0);

// affiche cette valeur
Serial.println(analogValue);
}
```

1.4.3.6. Fonctions propres au programme

Pour pouvoir utiliser une fonction propre à un programme, il faut au préalable la déclarer et comme son appel doit être possible à tout moment du déroulement du programme, il faut qu'elle soit déclarée de façon globale. C'est-à-dire que cela se fera en dehors des fonctions **setup()** et **loop()**.

Une fonction possède un nom, des paramètres d'entrée et des paramètres de sortie. On appelle **prototype d'une fonction**, la spécification de ces trois données :



Et tout ce que fait la fonction doit être placé entre deux accolades.

Il existe deux types de fonctions :

- Les fonctions qui ne renvoie aucune donnée,
- Les fonctions qui renvoie une donnée.

. Si la fonction ne renvoie aucune donnée, son type de retour sera **void** (vide).

Exemple :

Ce programme affiche le résultat de la division de 10 par 3 avec 1 à 5 chiffres après la virgule, dans le moniteur série.

Pour cela, on déclare une fonction appelée **affichfloat()** dont le type de retour est void (vide) et les paramètres d'entrées sont le nombre à virgule, **float a**, à afficher et un entier, b, représentant le nombre de chiffres après la virgule du nombre **a** qui seront affichés.

La fonction est appelée dans une boucle **for()** de la fonction setup() et le résultat de la division est affiché dans le moniteur série.

```
void affichfloat(float a, int b) { // Déclaration de la fonction « affichfloat »
    Serial.println(a,b); // Affichage de la variable a avec b chiffres après la virgule
}

void setup() {
    Serial.begin(9600);
    float result = 10.0/3.0;
    for(int i = 1 ; i < 6 ; i++){

   affichfloat(result,i);  // Appel de la fonction « affichfloat »

}
}

void loop() {
}
```

. Si la fonction renvoie une donnée, elle devra le faire avec le mot-clef **return**.

Exemple :

Le programme suivant affiche la table de multiplication de 2 dans le moniteur série. Pour cela, une fonction appelée **multiplication()** dont le type de retour est un entier **c** et les paramètres d'entrées sont 2 entiers **a** et **b** est déclarée.

La fonction est appelée dans une boucle **for()** de la fonction setup() et le résultat de la multiplication est affiché dans le moniteur série.

```
int multiplication (int a, int b) {  // Déclaration de la fonction « multiplication »

  int c ;
  c = a * b ;
  return c ;

}

void setup() {

Serial.begin(9600);

for(int i = 1 ; i < 10 ; i++){

   int result = multiplication(2,i);   // Appel de la fonction « multiplication »

   Serial.println(result);  // Affichage de la donnée de retour de la fonction « multiplication »

}
}

void loop() {

}
```

2. A propos d'ARDUINO LAB

2.1. ARDUINO LAB, Qu'est-ce que c'est ?

2.2. Installation et fonctionnement d'ARDUINO LAB

2.3. L'interface graphique - Les menus

2.4. Les bases de l'électronique

2.5. Les projets (étude de circuits) - Les activités

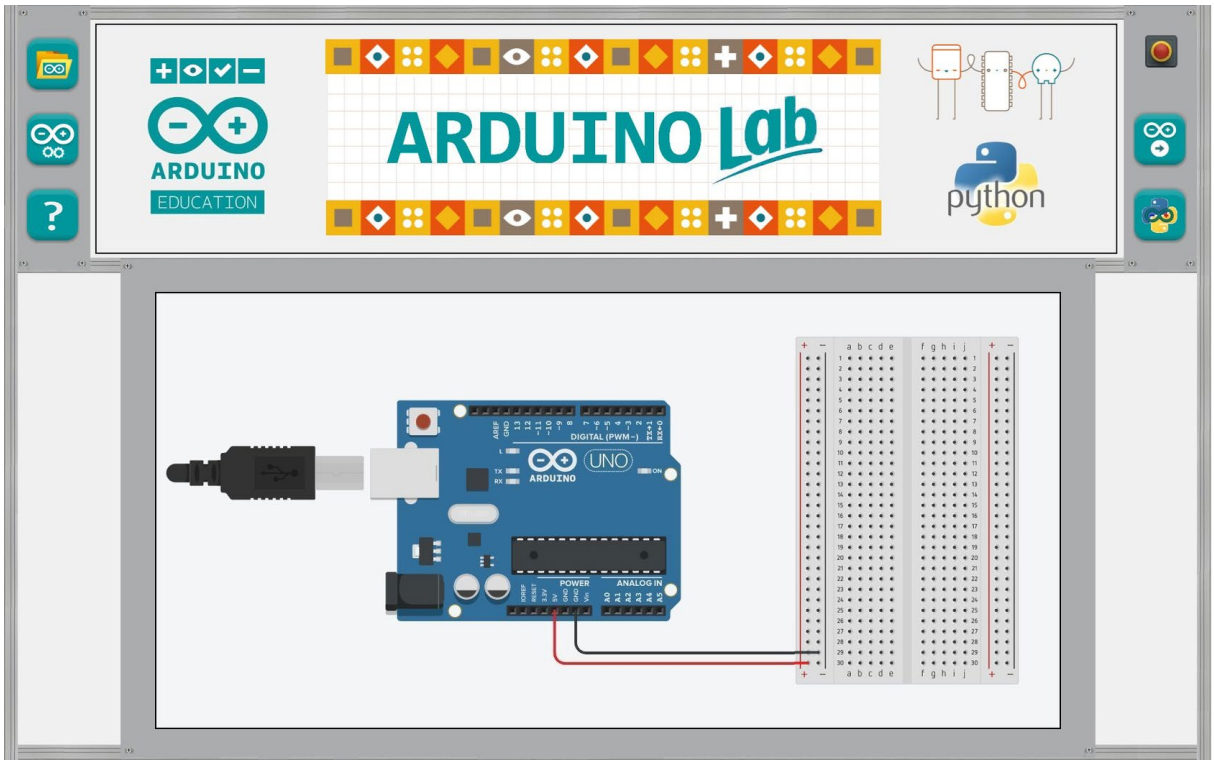
2.5.1 Premiers pas : DEL & Bouton poussoir

2.5.2 DEL RVB : Entrées et Sorties analogiques

2.5.3 Ondes sonores : Produire & Exploiter

2.5.4 Ondes ultrasonores : Vitesse & Distances

2.1. ARDUINO LAB, Qu'est-ce que c'est ?



Fenêtre d'accueil d'ARDUINO LAB

ARDUINO LAB est un logiciel programmé en Python 3 dans un but pédagogique pour permettre une découverte rapide et simple de l'**ARDUINO UNO** et de son utilisation dans le domaine des sciences.

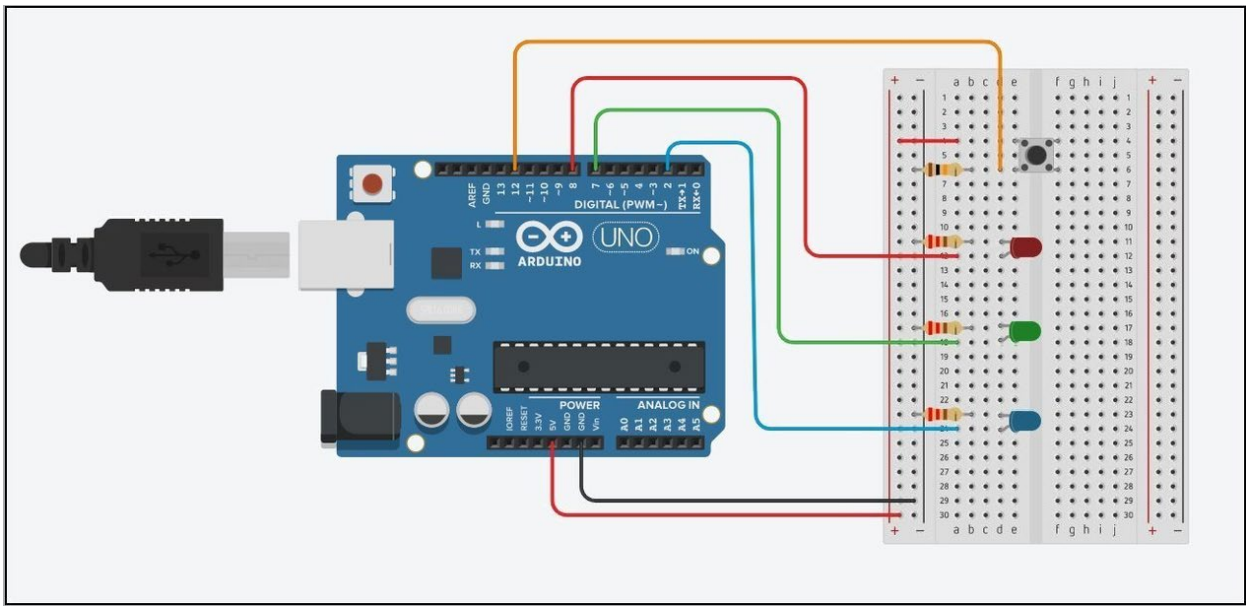
Si les cartes **Arduino** sont, de par leur conception, destinées à travailler de manière autonome, il est cependant intéressant de les utiliser comme interface physique sur un ordinateur pour piloter directement des matériels ou récupérer des informations issues de capteurs, à des fins de traitement et d'exploitations. Les données, qu'elles soient des commandes ou des informations, transiteront par la connexion USB.

Pour cela, deux programmes sont nécessaires :

. Un programme "donneur d'ordre" sur l'ordinateur,

. et un "pilote", animant le microcontrôleur, qui comme son nom l'indique, pilotera les matériels en réponse aux ordres reçus et fournira des données en retour.

ARDUINO LAB est le programme "donneur d'ordre" qui permet de contrôler l'ARDUINO graphiquement, que ce soit en entrée ou en sortie par l'intermédiaire de circuits électroniques déjà conçus :



Exemple de circuits étudiés dans ARDUINO LAB

La liaison entre **ARDUINO LAB** et l'Arduino fonctionne dans les deux sens. Toute interaction sur l'Arduino (par exemple, l'appui sur un bouton poussoir) est visible sur l'interface graphique **d'ARDUINO LAB**.

Un mode "Simulation" permet de tester les circuits avant d'effectuer la liaison avec l'Arduino.

Avec **ARDUINO LAB**, Il est également possible de contrôler chaque broche de la platine Arduino pour, par exemple, avec une entrée reliée à un potentiomètre, piloter l'intensité d'une DEL ou l'angle d'un servomoteur…

Enfin, **ARDUINO LAB** permet aussi d'enregistrer les données d'un capteur et d'en effectuer leurs exploitations, fonctionnalité très intéressante dans le domaine des sciences.

2.2. Installation et fonctionnement d'ARDUINO LAB

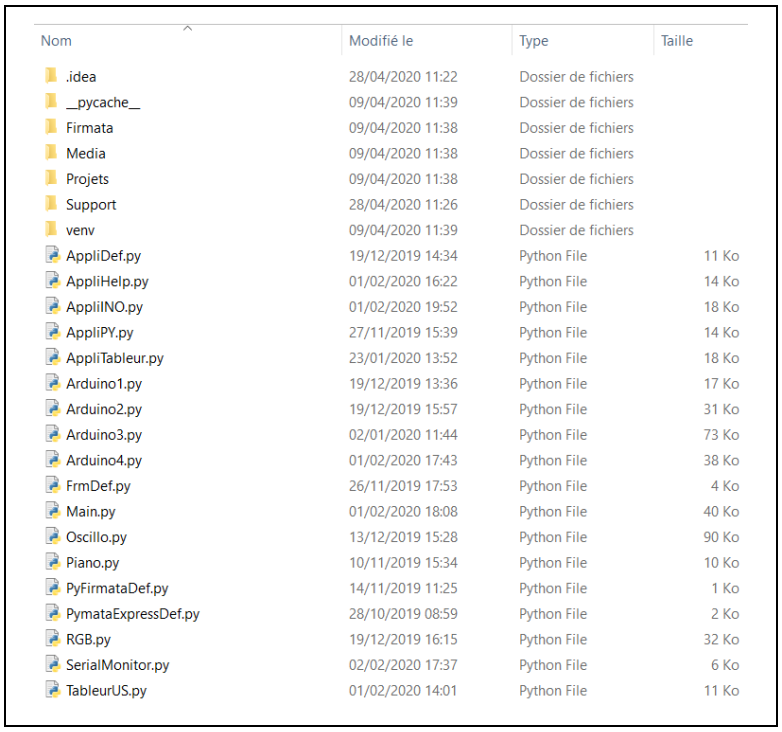
. Installation d'ARDUINO LAB

Python étant un langage de programmation interprété, pour faire fonctionner **ARDUINO LAB**, il faut qu'un interpréteur **Python** soit installé.

ARDUINO LAB nécessite au minimum la version 3.7 de **Python** et l'installation des bibliothèques suivantes :

- **pyfirmata** (pour contrôler l'Arduino selon le protocole **Firmata Standard**)
- **pymata-express** (pour contrôler l'Arduino selon le protocole **Firmata Express)**
- **asyncio** (pour la programmation asynchrone)
- **matplotlib** (pour tracer et visualiser des données sous formes de graphiques)
- **numpy** (pour le calcul scientifique)
- **Pillow** (pour le traitement des images)
- **scipy** (pour le traitement des données)
- **PyMuPDF** (pour l'affichage des documents PDF)

Une fois l'environnement de travail configuré et après avoir téléchargé puis décompressé le fichier **"ArduinoLab.zip"** qui contient tous les fichiers et dossiers nécessaire à son fonctionnement, **ARDUINO LAB** est démarré à l'aide du fichier **"Main.py"** situé dans le dossier principal **"ArduinoLab"** du programme :



Nom	Modifié le	Type	Taille
.idea	28/04/2020 11:22	Dossier de fichiers	
__pycache__	09/04/2020 11:39	Dossier de fichiers	
Firmata	09/04/2020 11:38	Dossier de fichiers	
Media	09/04/2020 11:38	Dossier de fichiers	
Projets	09/04/2020 11:38	Dossier de fichiers	
Support	28/04/2020 11:26	Dossier de fichiers	
venv	09/04/2020 11:39	Dossier de fichiers	
AppliDef.py	19/12/2019 14:34	Python File	11 Ko
AppliHelp.py	01/02/2020 16:22	Python File	14 Ko
AppliINO.py	01/02/2020 19:52	Python File	18 Ko
AppliPY.py	27/11/2019 15:39	Python File	14 Ko
AppliTableur.py	23/01/2020 13:52	Python File	18 Ko
Arduino1.py	19/12/2019 13:36	Python File	17 Ko
Arduino2.py	19/12/2019 15:57	Python File	31 Ko
Arduino3.py	02/01/2020 11:44	Python File	73 Ko
Arduino4.py	01/02/2020 17:43	Python File	38 Ko
FrmDef.py	26/11/2019 17:53	Python File	4 Ko
Main.py	01/02/2020 18:08	Python File	40 Ko
Oscillo.py	13/12/2019 15:28	Python File	90 Ko
Piano.py	10/11/2019 15:34	Python File	10 Ko
PyFirmataDef.py	14/11/2019 11:25	Python File	1 Ko
PymataExpressDef.py	28/10/2019 08:59	Python File	2 Ko
RGB.py	19/12/2019 16:15	Python File	32 Ko
SerialMonitor.py	02/02/2020 17:37	Python File	6 Ko
TableurUS.py	01/02/2020 14:01	Python File	11 Ko

Attention :

L'emplacement du dossier décompressé **"ArduinoLab"** n'a pas d'importance, mais tous les fichiers et dossiers contenus dans ce dossier ne doivent en aucun cas être modifiés ou déplacés.

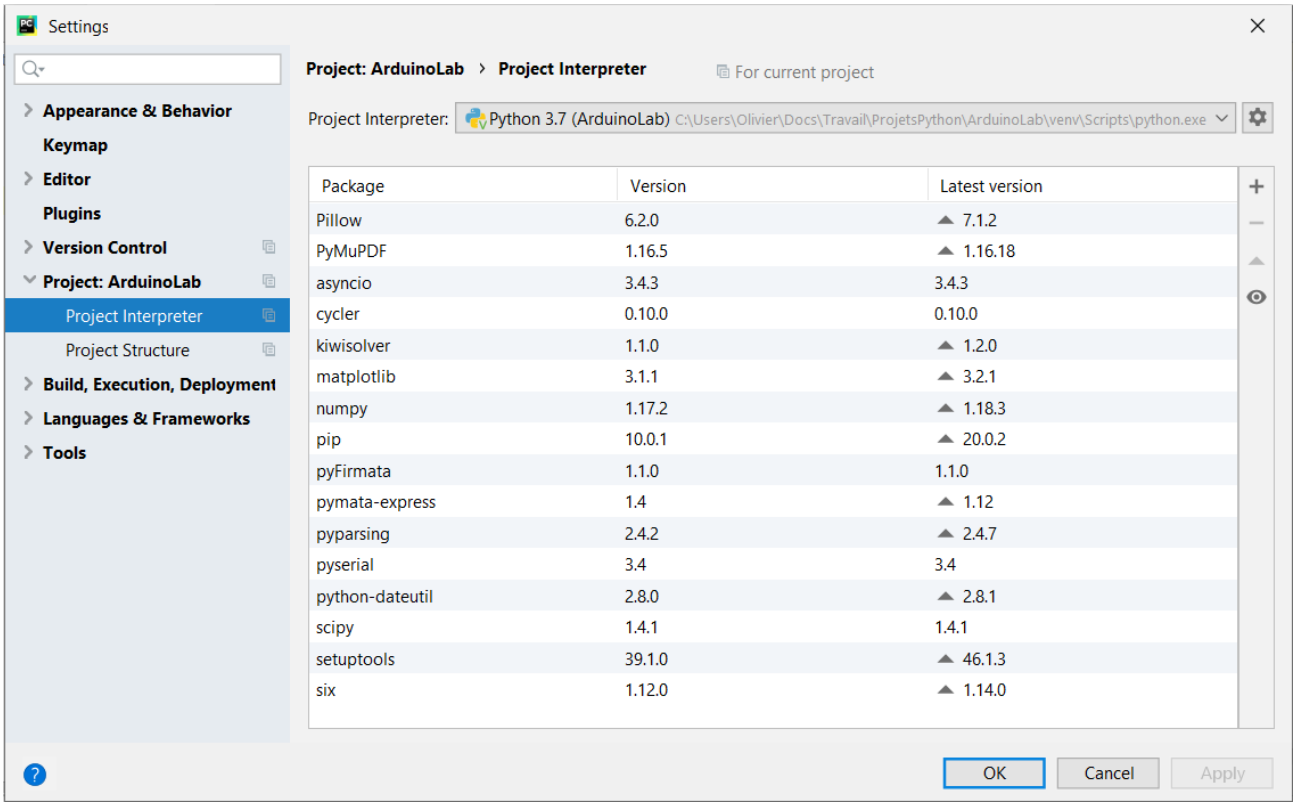
Remarques :

Le dossier principal **"ArduinoLab"** du programme contient un dossier nommé **"venv"** dans lequel se situe un environnement virtuel de programmation avec les bibliothèques indispensables citées ci-dessus.

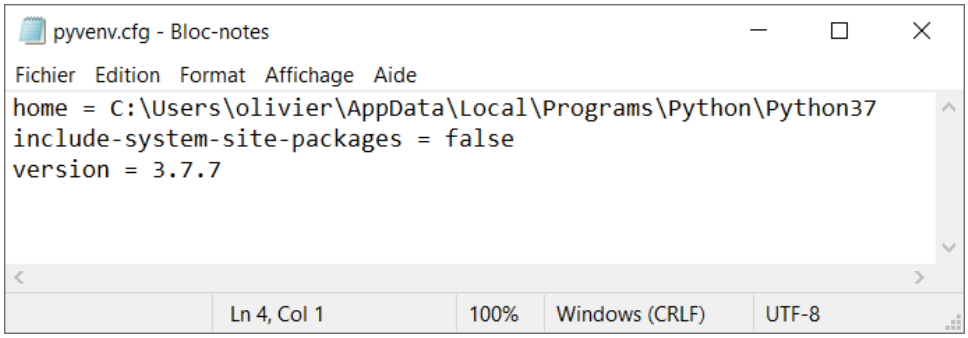
Cet environnement de programmation peut être utilisé comme interpréteur du programme.
Dans ce cas, même si l'installation d'une distribution **Python** (3.7 au minimum) est indispensable, les bibliothèques dont **ARDUINO LAB** dépendent n'auront pas à être ajoutées à la distribution originale installée.

L'utilisation de l'environnement de programmation virtuel pour le fonctionnement d'**ARDUINO LAB** se configure par l'intermédiaire d'un environnement de développement Python (IDE), par exemple, **PyCharm**.

Ainsi dans **PyCharm**, après avoir ouvert le dossier d'**ARDUINO LAB**, il suffit d'indiquer dans les réglages que l'environnement virtuel est l'interpréteur du projet **:**



Il faudra cependant au préalable modifier le fichier ”**pyvenv.cfg**” situé dans le dossier **”ArduinoLab/venv/”** pour indiquer le chemin d’installation de la distribution Python :



. Pré-requis au fonctionnement d'ARDUINO LAB

Pour fonctionner, **ARDUINO LAB,** nécessite l’installation d’un programme "pilote" dans la mémoire de l’Arduino (sauf en mode "Simulation" bien-sûr) pour envoyer des ordres à l’Arduino ou recevoir des données via le port USB.

Suivant les circuits étudiés ou les capteurs utilisés, **ARDUINO LAB** utilise deux protocoles de communication avec l’Arduino différents, "**Firmata standard**" ou "**Firmata express**".

Le chargement, dans la mémoire de l'Arduino, du code "pilote" doit être fait soit manuellement en utilisant le logiciel "**IDE ARDUINO**", soit par l'intermédiaire du menu "**Paramètres**" d'**ARDUINO LAB**. Un mode de chargement automatique du protocole de communication en fonction du circuit étudié est également disponible dans ce menu.

Quoi qu’il en soit, pour utiliser toutes les fonctionnalités d’**ARDUINO LAB,** notamment le téléversement de codes écrits en langage "Arduino", il est nécessaire d’installer, sur l’ordinateur utilisé pour contrôler l’Arduino, le logiciel "**IDE ARDUINO**" qui est disponible à l’adresse suivante :

https://www.arduino.cc/en/Main/Software

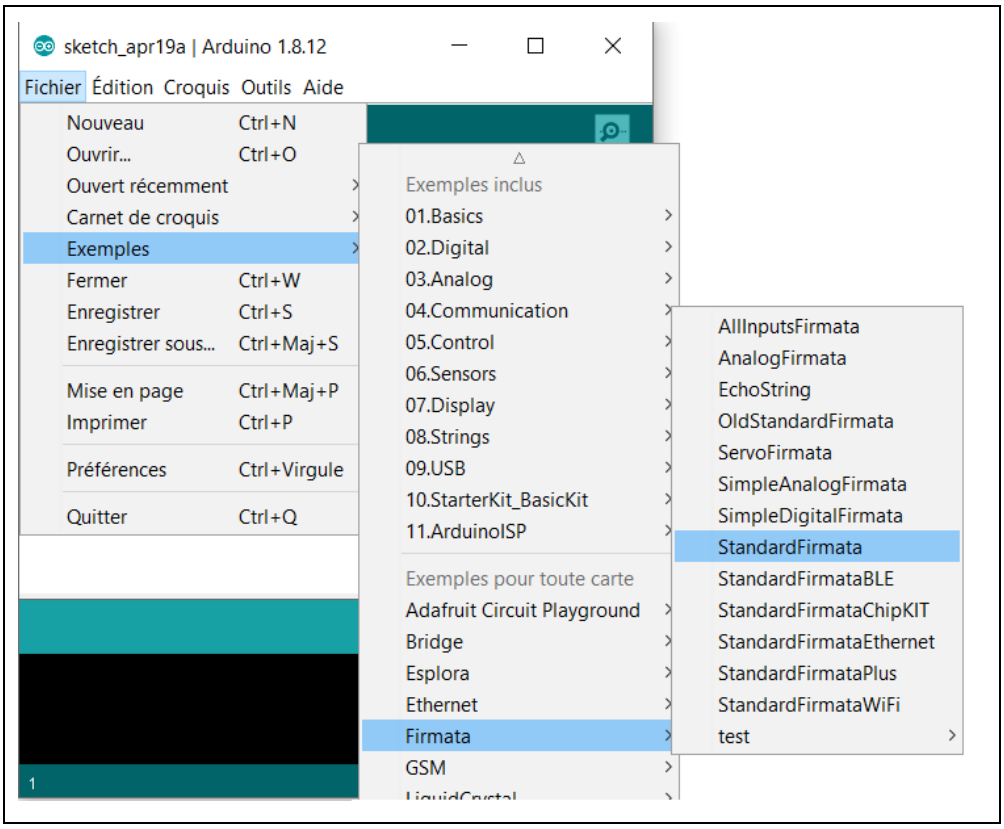
. **Chargement manuel du code "Firmata Standard"** :

- Brancher l’Arduino via un port USB,

- Afin de charger la librairie "**Firmata standard**" sur l’ARDUINO, il faut lancer le logiciel "**IDE ARDUINO**", puis sélectionner :

Fichier > Exemples > Firmata > Standard Firmata,

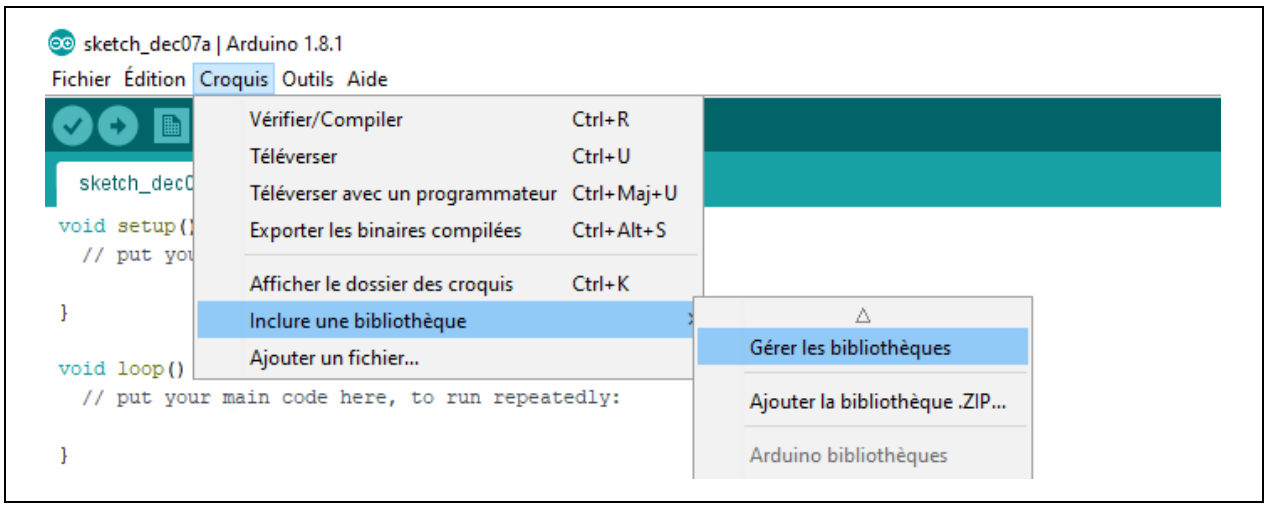
- puis cliquer sur "**téléverser**".



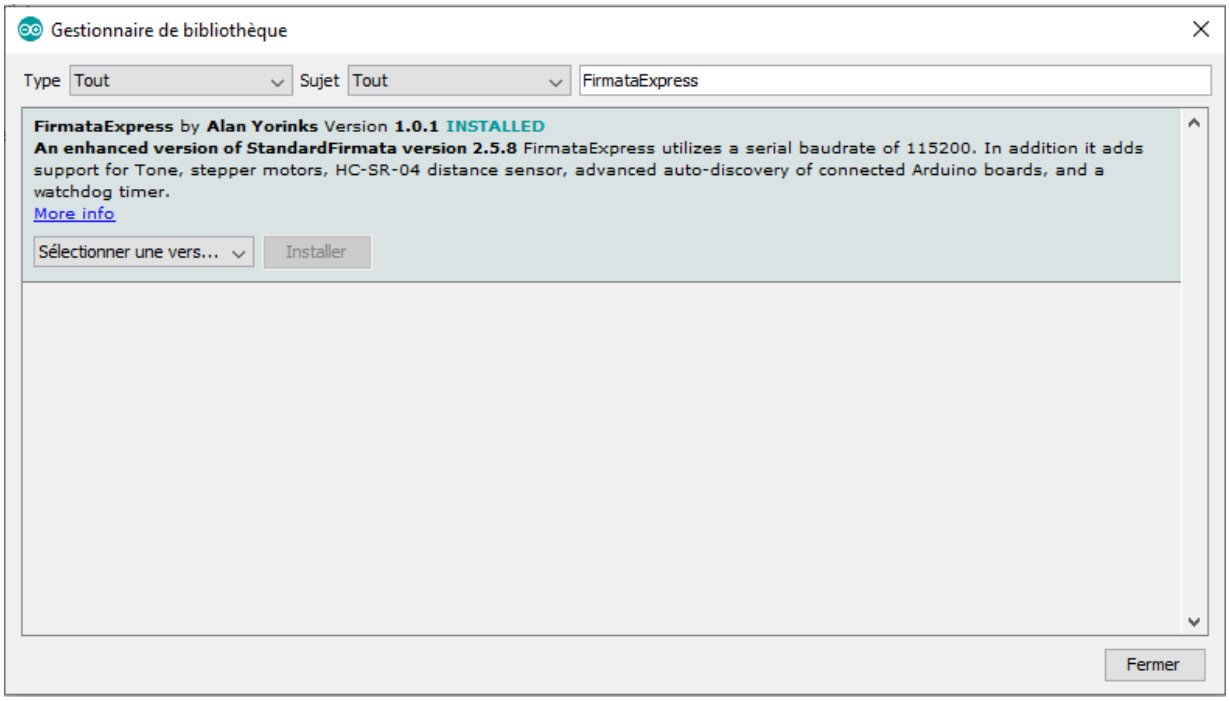
. **<u>Chargement manuel du code "Firmata Express"</u>** :

Par défaut, la bibliothèque **"Firmata Express"** n'est pas incluse dans l'"**IDE Arduino".** Il faut donc procéder à son installation :

- Ouvrir le logiciel "**IDE ARDUINO**",
- Sélectionner "**Croquis/Inclure une bibliothèque/Gérer les bibliothèques**",



- Entrer "**FirmataExpress**" dans la zone de recherche :



- et cliquer sur "**installer**".

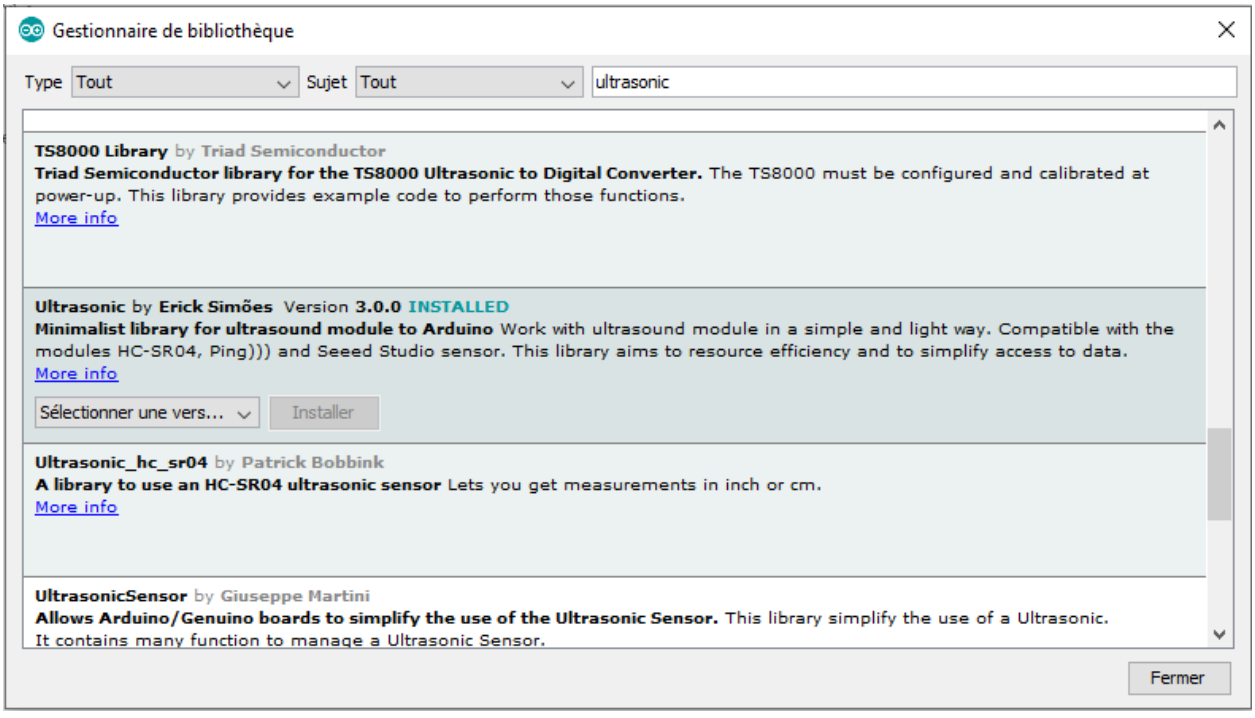
"**FirmataExpress**" nécessite également que la librairie "**Ultrasonic by Erick Simões**" soit installée.

De même que précédemment, en utilisant le logiciel Arduino IDE :

- Sélectionner "**Croquis/Inclure une bibliothèque/Gérer les bibliothèques**",

- Entrer "**Ultrasonic**" dans la zone de recherche,

- Sélectionner "**Ultrasonic by Erick Simões**"



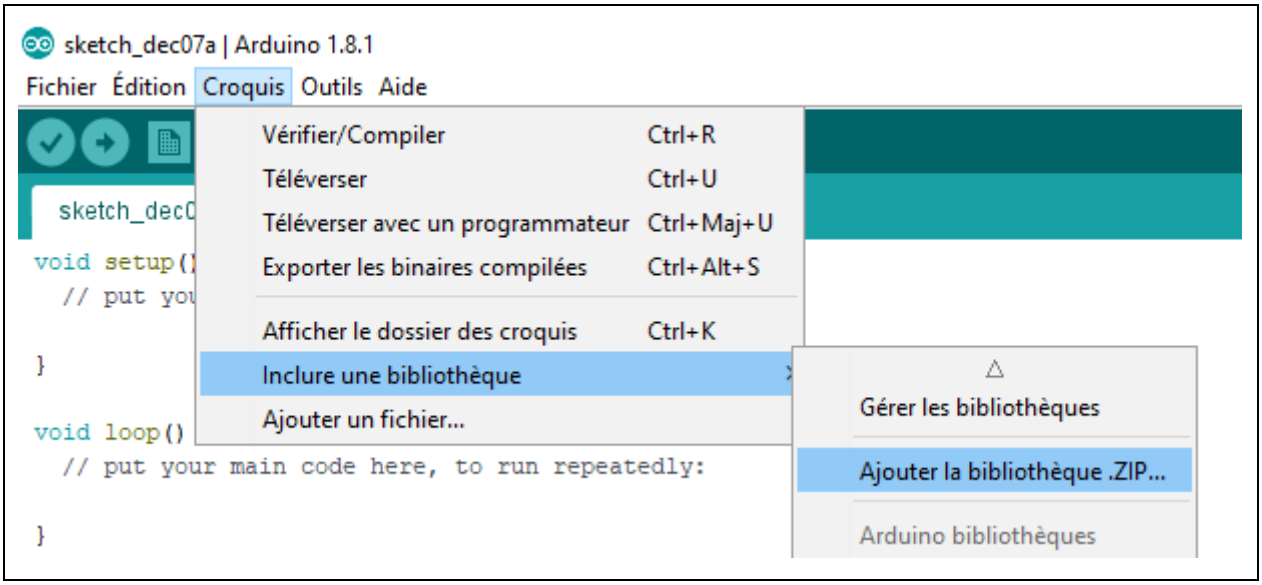
- et cliquer sur "**installer**".

Sans connexion internet, les librairies peuvent être installées à partir des fichiers "**zip**" présents dans le dossier "**Support/Librairies Arduino**" du répertoire d'installation d'**ARDUINO LAB** :

- Ouvrir le logiciel "**IDE ARDUINO**",

- Sélectionner "**Croquis/Inclure une bibliothèque/Ajouter la bibliothèque .ZIP**",

- Sélectionner le fichier .ZIP de la librairie à installer.



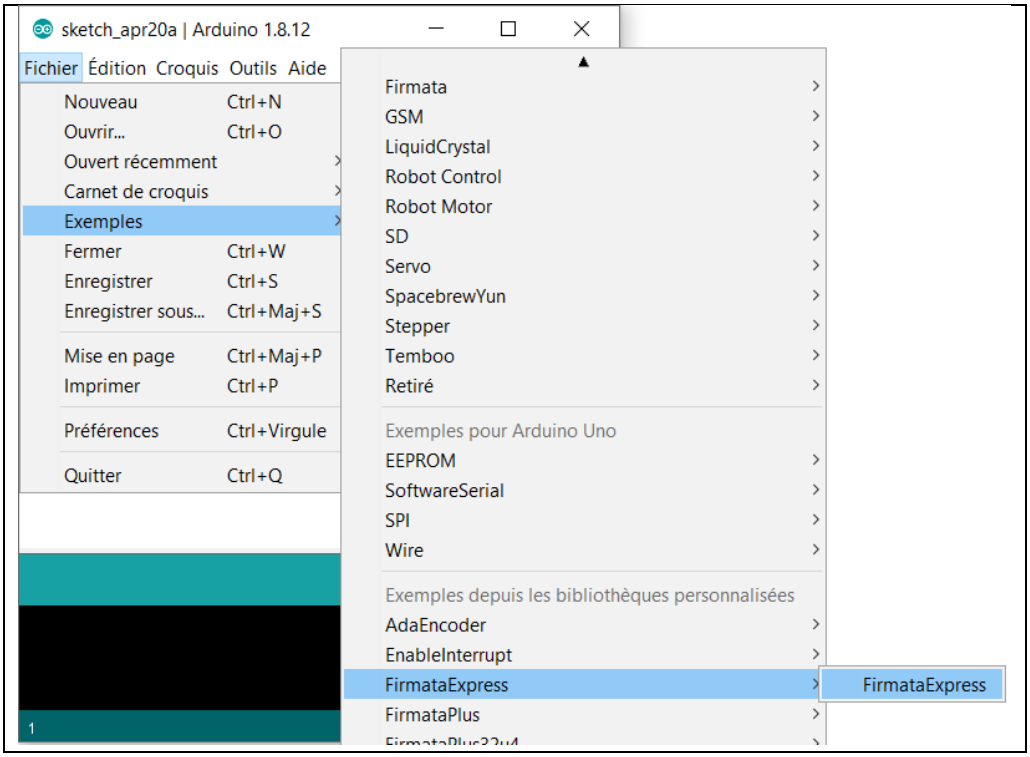
Le chargement du code "**Firmata Express**" dans la mémoire de l'Arduino est maintenant possible :

- Brancher l'Arduino via un port USB,

- Afin de charger la librairie "Firmata express" sur l'ARDUINO, il faut lancer le logiciel "IDE ARDUINO", puis sélectionner :

Fichier > Exemples > FirmataExpress > FirmataExpress,

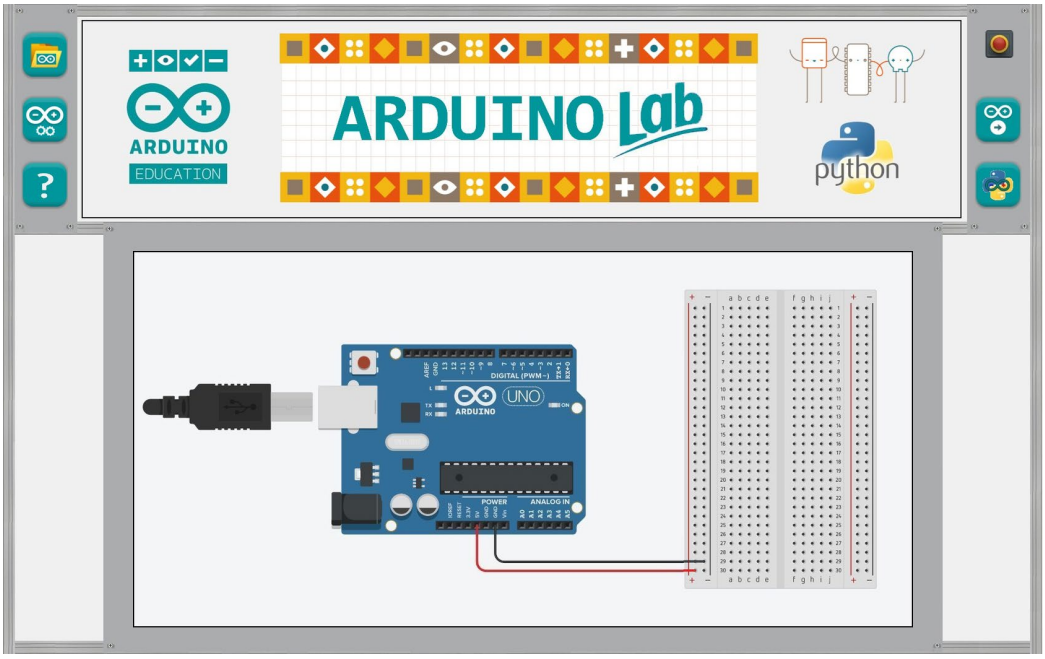
- puis cliquer sur "téléverser".



A partir de là vous n’avez plus besoin du logiciel **Arduino IDE**, vous pourrez contrôler votre Arduino uniquement à partir d’**ARDUINO LAB**.

2.3. L'interface graphique - Les menus

A partir de la fenêtre d'accueil, on accède aux menus d'**ARDUINO LAB :**



2.3.1 Menu "Paramètres"

Pour accéder au menu "**Paramètres**", il faut cliquer sur :

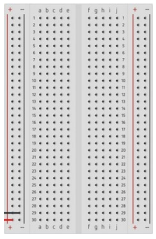
Le Menu "**Paramètres**" d'**ARDUINO LAB** permet d'indiquer:

- le mode de fonctionnement du logiciel, à savoir : "**Contrôle de l'Arduino**" ou "**Simulation**",
- le port "**COM**" sur lequel l'Arduino est connecté (si un seul Arduino est connecté, le port "**COM**" est sélectionné automatiquement)
- le mode de sélection du protocole de communication entre le logiciel et l'Arduino (sélection manuelle ou automatique, chargement de "**Firmata standard**" ou de "**Firmata Express**"),
- l'emplacement du dossier d'installation du logiciel "**Arduino IDE**".

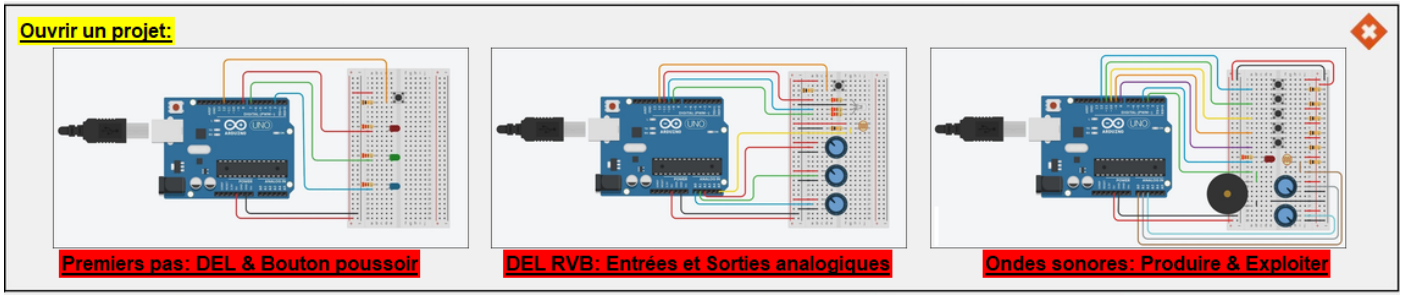
2.3.2 Menu "Ouvrir un projet"

Pour ouvrir un projet et étudier un circuit, il faut cliquer sur le bouton :

Ou sur la plaque d'essais :

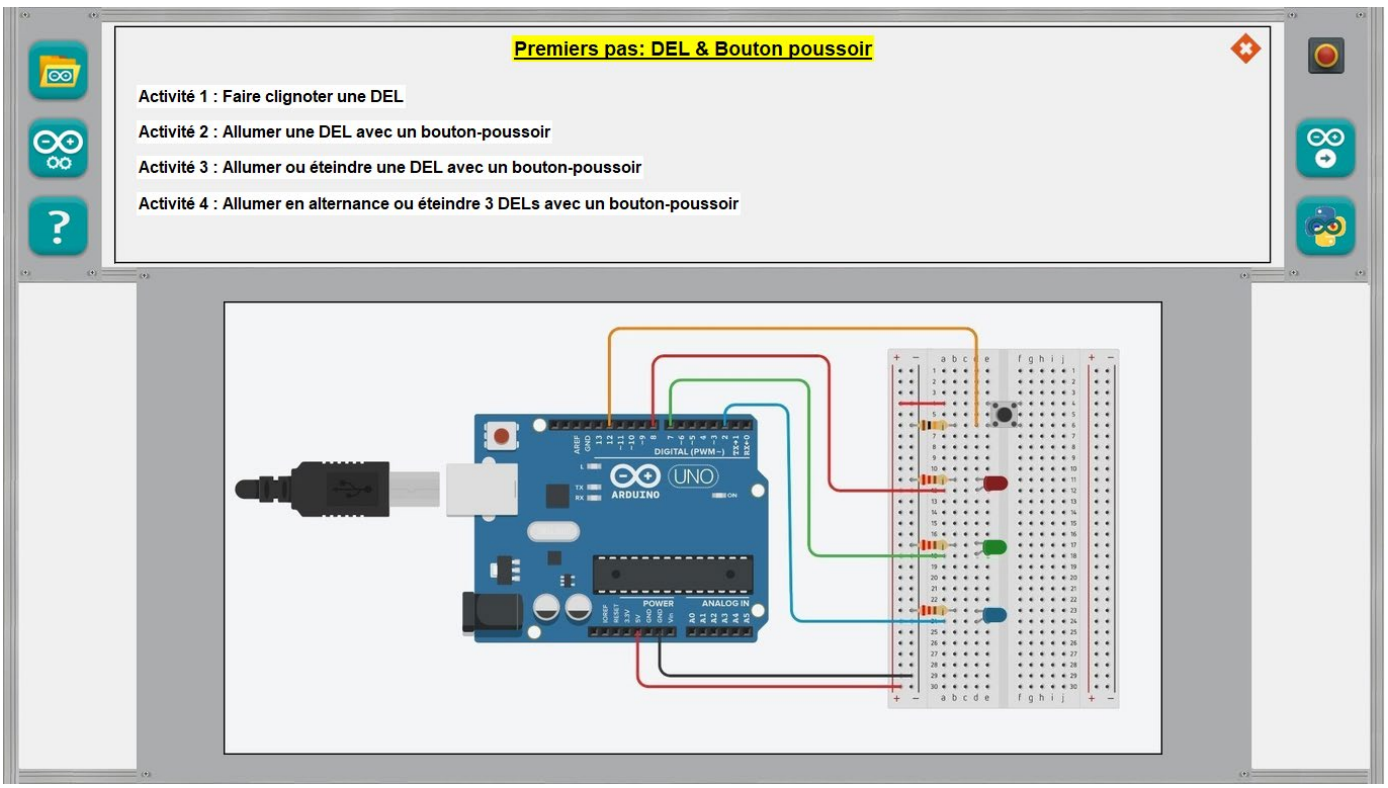


Et choisir le circuit à étudier dans la fenêtre qui s'ouvre, en cliquant sur son image :

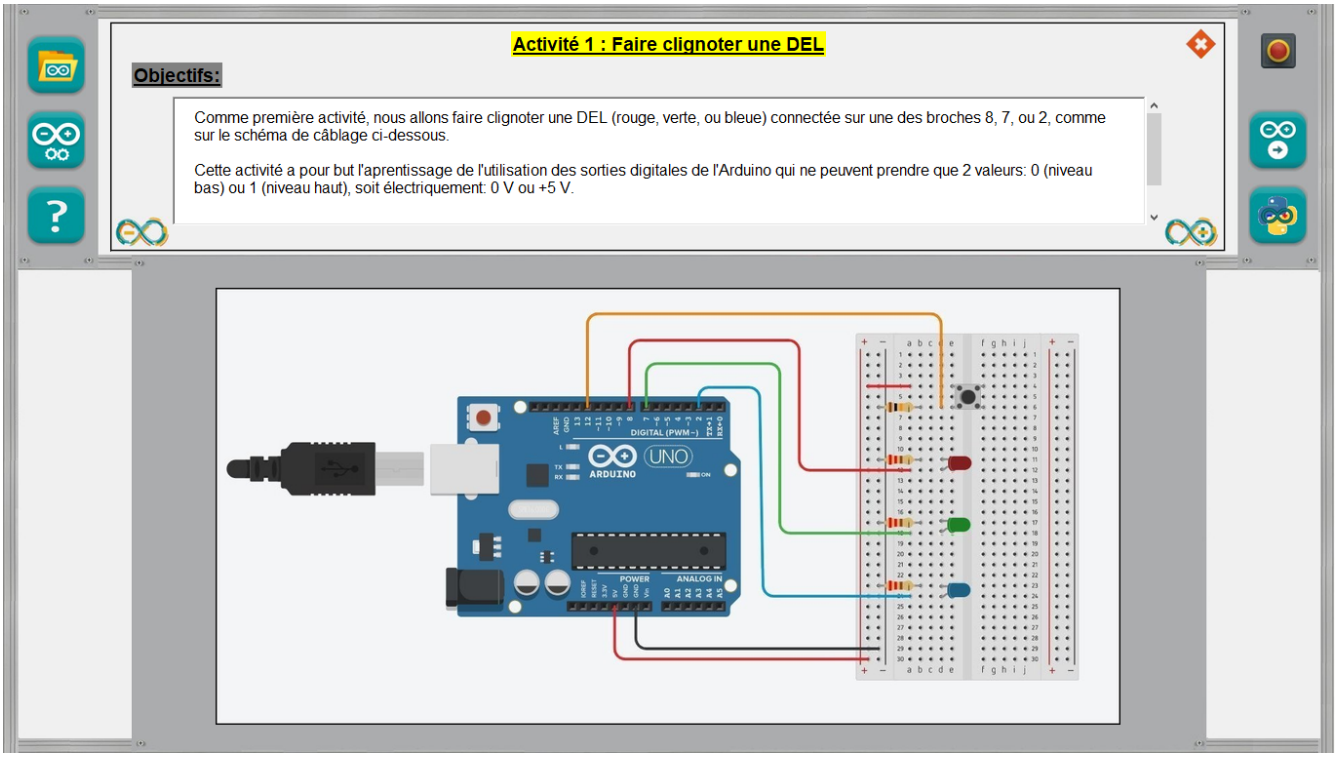


2.3.3 Menu "Sélectionner une activité"

Après avoir choisi un circuit à étudier, une fenêtre avec le circuit et la liste des activités liées au circuit sélectionné s'affiche :

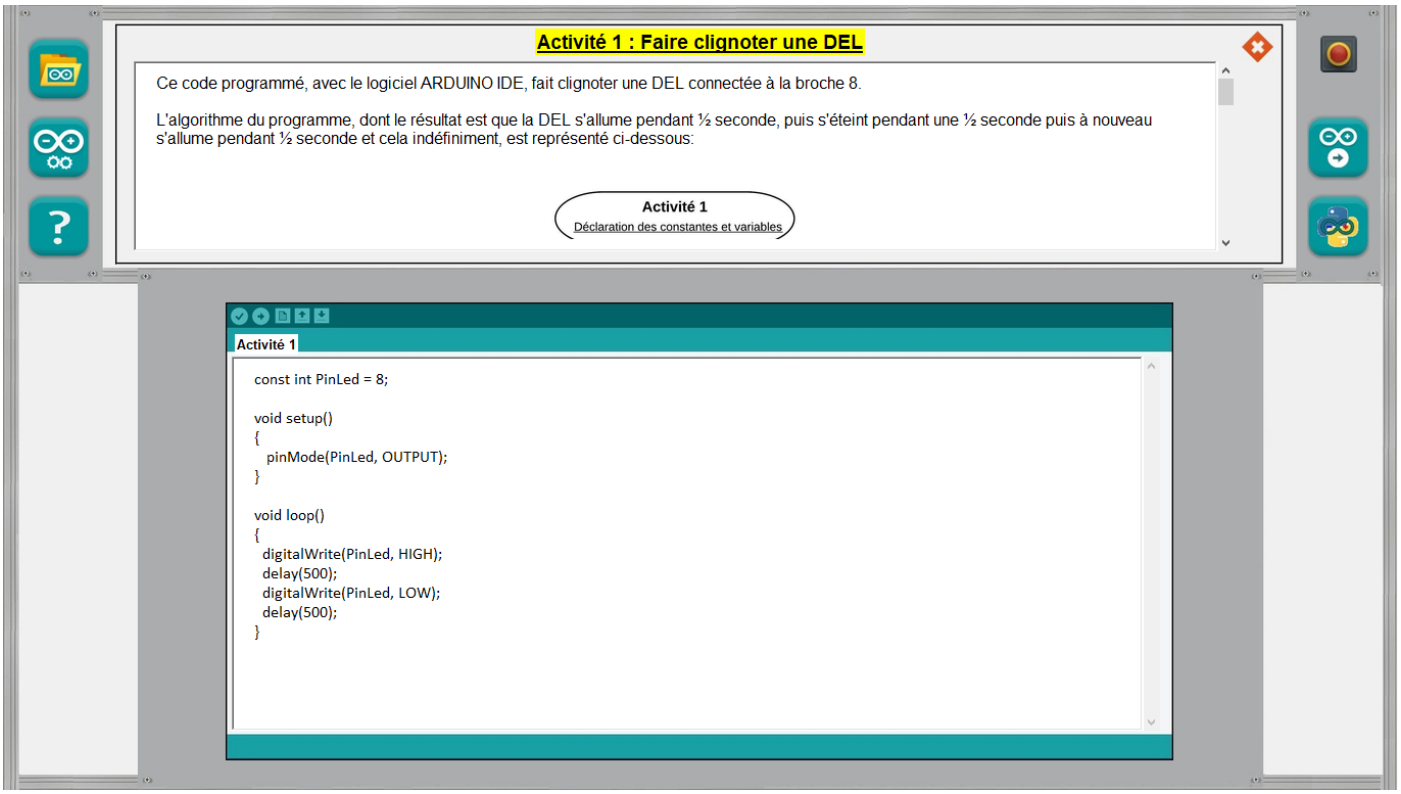


Il suffit de cliquer sur une des activités pour que la fenêtre correspondant à l'activité choisie s'ouvre et affiche sa description :



2.3.4 Menu "Etude du code en langage ARDUINO IDE"

Le code en langage **ARDUINO IDE** permettant de réaliser l'activité choisie est affiché en cliquant sur le bouton :



L'accès à ce menu de l'interface graphique n'est possible que si une activité est sélectionnée.

Sur la partie supérieure de la fenêtre, une description et l'algorithme du code sont affichés et en dessous, on retrouve le code de l'activité que l'on peut modifier.

Il est possible de sauvegarder le code modifié, d'éditer un nouveau code et d'ouvrir un code préalablement enregistré.

Le code original ou modifié après sauvegarde peut être vérifié et téléversé directement depuis **ARDUINO LAB** dans l'Arduino.

Dans ce cas, l'Arduino est dans un fonctionnement autonome et il n'y a plus d'interaction possible avec **ARDUINO LAB**. Pour réutiliser toutes les fonctions d'**ARDUINO LAB**, il faudra téléverser le protocole de communication adéquate (Firmata standard ou express).

Les fonctionnalités décrites ci-dessus sont accessibles en cliquant sur les boutons :

- Vérifier un code :
- Téléverser un code :
- Editer un nouveau code :
- Ouvrir un code :
- Sauvegarder un code :

2.3.5 Menu "Etude du code en Python"

Le code en Python permettant de réaliser l'activité choisie est affiché en cliquant sur le bouton :

Comme pour le menu précédant, L'accès à ce menu de l'interface graphique n'est possible que si une activité est sélectionnée.

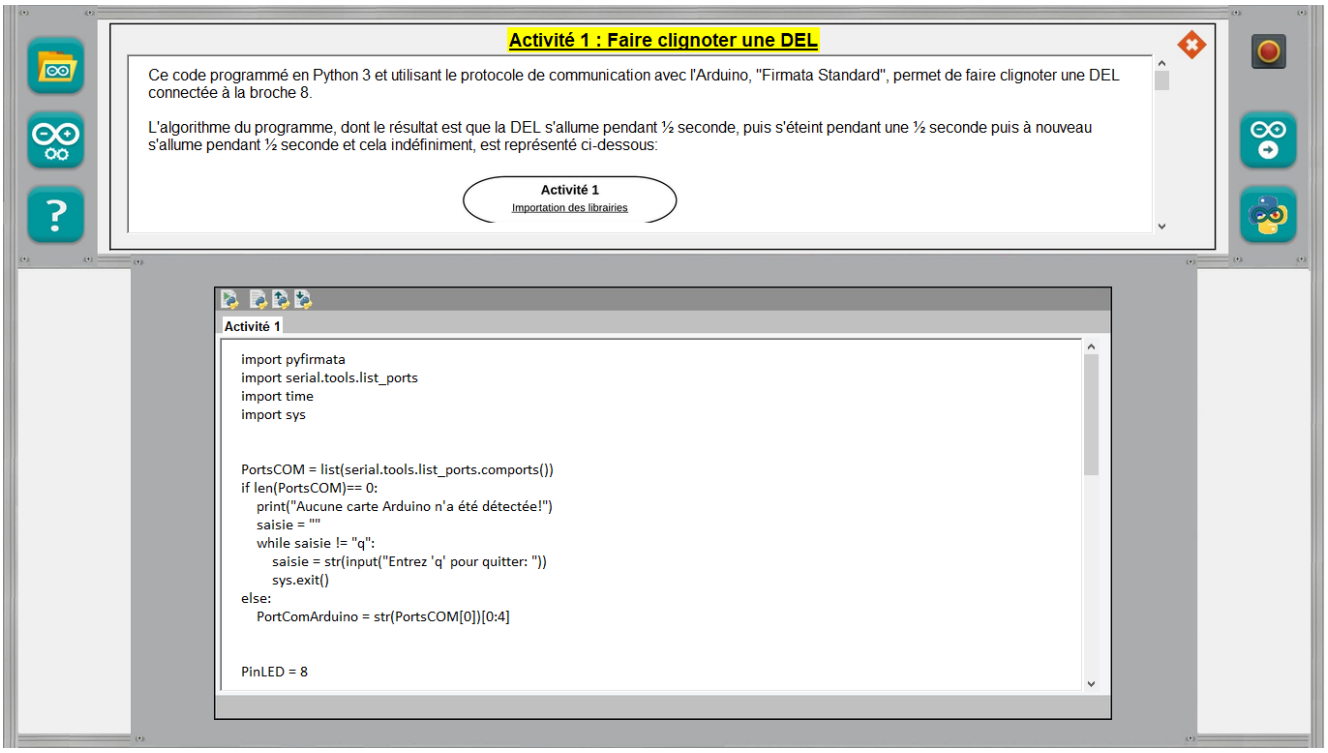
De même, sur la partie supérieure de la fenêtre, une description et l'algorithme du code en Python sont affichés et en dessous, on retrouve le code de l'activité que l'on peut modifier.

Il est également possible de sauvegarder le code modifié, d'éditer un nouveau code et d'ouvrir un code préalablement enregistré.

Le code original ou modifié après sauvegarde est exécutable. Cependant, il faudra s'assurer que le bon protocole de communication entre Python et l'Arduino a bien été téléverser.

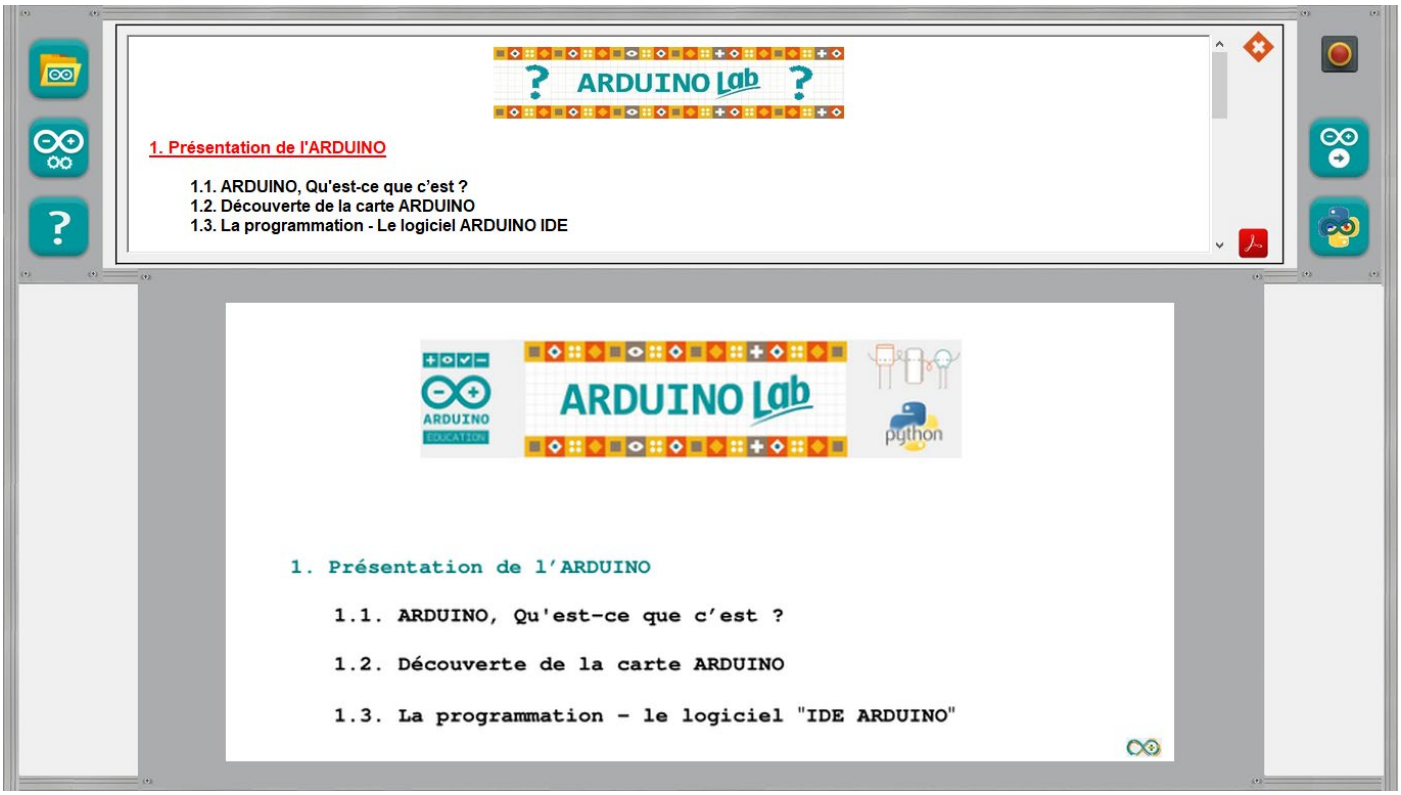
Les fonctionnalités décrites ci-dessus sont accessibles en cliquant sur les boutons :

- Exécuter un code : 
- Ouvrir un code : 
- Editer un nouveau code : 
- Sauvegarder un code : 



2.3.6 Menu "Aide"

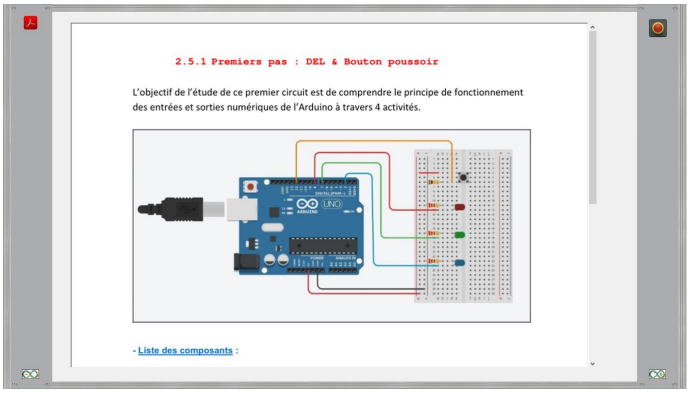
A tout moment, une aide contextuelle est affichée en cliquant sur le bouton : 

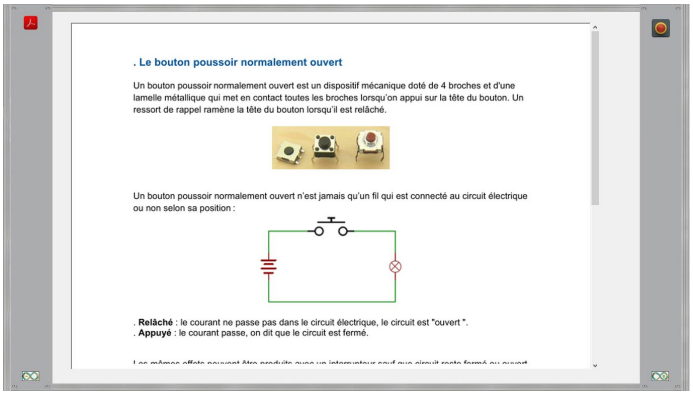


L'accès à l'aide générale d'**ARDUINOLAB** se fait à partir de la fenêtre d'accueil**.**

Quand un circuit ou une activité est sélectionné, l'aide affichée est spécifique à la sélection en cours.

Un clic droit sur certains éléments des circuits ou des fenêtres d'exploitation affiche également une aide contextuelle sur les composants, les données reçues…





<u>Remarques</u> :

. La fenêtre principale et les fenêtres secondaires d'ARDUINO LAB sont fermées en cliquant sur: 

. Quelle que soit la résolution de l'écran, les fenêtres et les objets d'Arduino LAB sont redimensionnés de façon à occuper tout l'espace disponible.

Cependant, l'affichage est optimisé pour les résolutions d'écran suivantes :

. 1366 x 768

. 1680 x 1050 (et toutes les résolutions 8/5)

. 1920 x 1080 (et toutes les résolutions 16/9)

2.4. Les bases de l'électronique

Avant de voir en détail les différents circuits étudiés, nous allons faire quelques rappels sur l'électronique :

. Petit rappel sur l'électricité

L'électricité est un déplacement d'électrons dans un milieu conducteur.
Pour que ces électrons se déplacent tous dans un même sens, il faut qu'il y ait une différence du nombre d'électrons entre les deux extrémités du circuit électrique.
Pour maintenir cette différence du nombre d'électrons, on utilise un générateur (pile, accumulateur, alternateur...)

La différence de quantité d'électrons entre deux parties d'un circuit s'appelle la **différence de potentiel** et elle se mesure en **Volts (V)**et se note *U.*

Le débit d'électrons dans le conducteur correspond à *l'intensité*, aussi appelée *courant*. Elle se mesure en *Ampères* (A) et se note *I.*

La puissance électrique se note *P* et se mesure en *Watts* (W). Elle exprime la quantité de courant (I), transformée en chaleur ou en mouvement.

La puissance P est le produit de la tension U et de l'intensité I.

$$P = U \cdot I$$

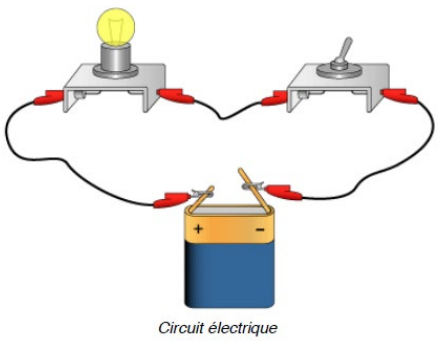
Le circuit électrique

Un circuit électrique est une association de dipôles (générateur, résistances, ampoules, …) reliés par des conducteurs.

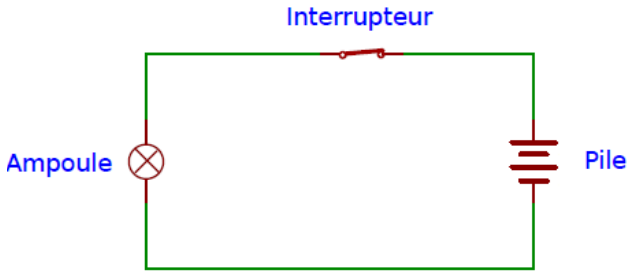
Pour qu'un courant circule, il faut que le circuit soit fermé et qu'il contienne un générateur (une pile par exemple). Il circule du pôle (+) au pôle (-) du générateur.

Une pile est constituée d'un milieu contenant de nombreux électrons en trop, et d'un second milieu en manque d'électrons. Quand on relie les deux pôles de la pile (le + et le -) avec un fil électrique (le conducteur), les électrons vont alors se déplacer du milieu riche en électrons vers le milieu pauvre.

Si on place une lampe électrique entre les deux, le passage des électrons va générer de la lumière.



Circuit électrique

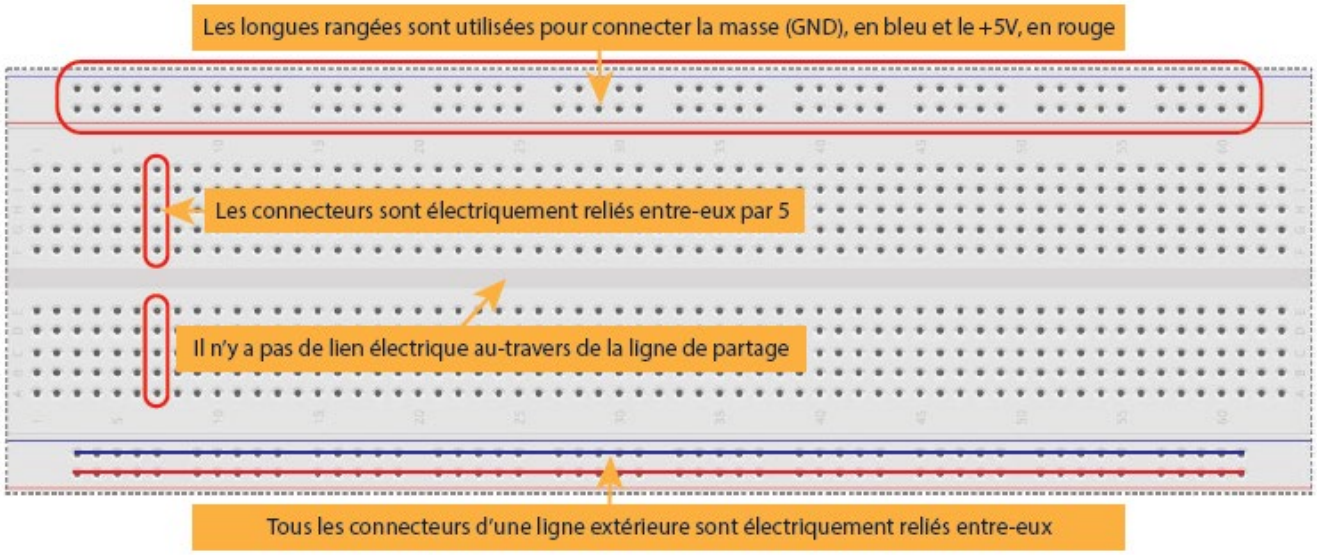


Dans le circuit ci-dessus, lorsque l'interrupteur est enclenché, on dit que le circuit est fermé. Les électrons vont alors se déplacer et l'énergie de ce déplacement pourra être exploitée pour allumer une lampe ou faire fonctionner un moteur, par exemple.
Lorsque l'interrupteur est déclenché, on dit que le circuit est ouvert. Le pôle positif n'étant alors plus relié au pôle négatif, les électrons ne peuvent plus se déplacer.

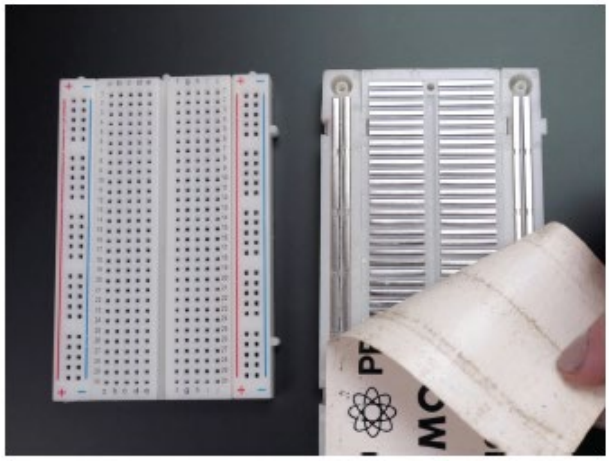
Dans les circuits électriques que nous allons étudier, L'Arduino servira d'alimentation électrique du circuit, comme une pile.

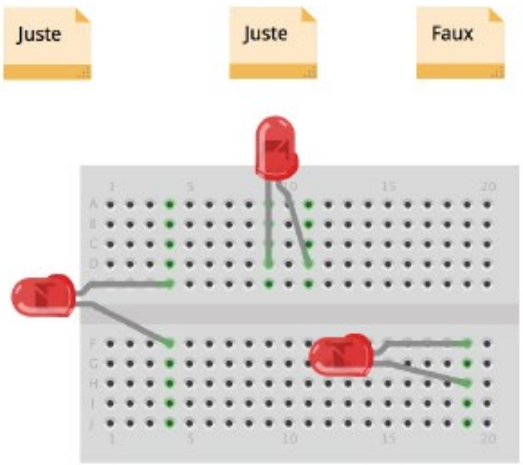
La platine d'expérimentation

Une platine d'expérimentation (appelée breadboard) permet de réaliser des prototypes de montages électroniques sans soudure et donc de pouvoir réutiliser les composants.



Tous les connecteurs dans une rangée de 5 sont reliés entre eux. Donc si on branche deux éléments dans un groupe de cinq connecteurs, ils seront reliés entre eux. Il en est de même des alignements de connecteurs rouges (pour l'alimentation) et bleus (pour la terre).





Les composants doivent ainsi être placés à cheval sur des connecteurs qui n'ont pas de liens électriques entre eux.

Les résistances

Une **résistance** est un composant électronique ou électrique dont la principale caractéristique est d'opposer une plus ou moins grande résistance (mesurée en ohms : Ω) à la circulation du courant électrique.

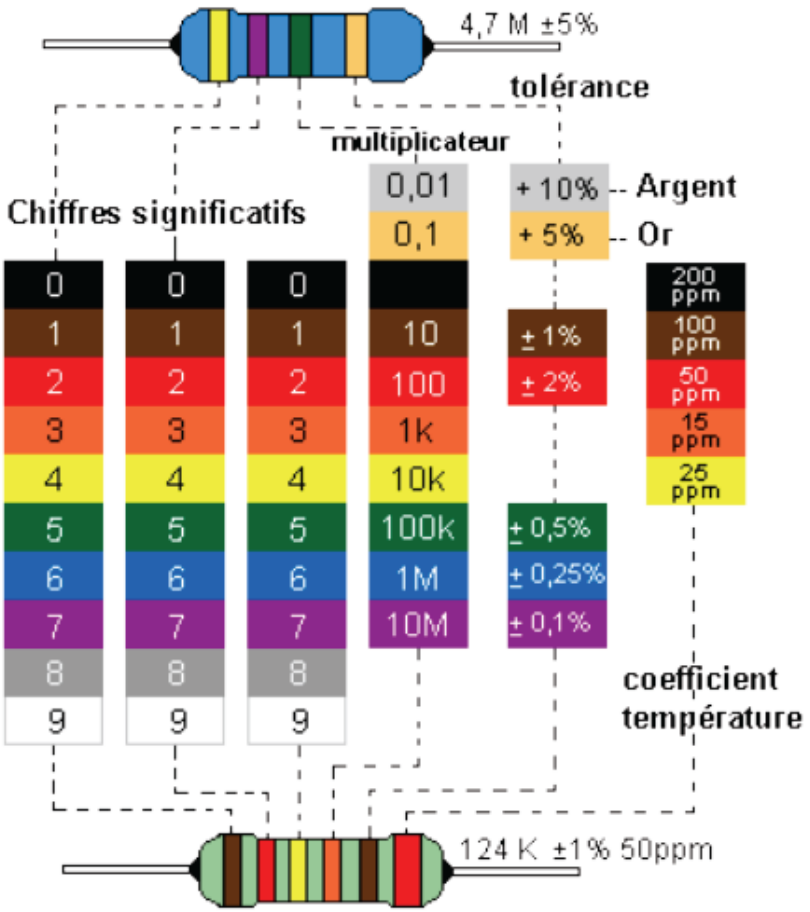
Ainsi, pour une tension fixe, plus la résistance est faible, plus le courant la traversant est fort. Cette proportion est vérifiée par la loi d'Ohm :

$$U = R \times I \text{ et donc } R = U/I \text{ et } I = U/R$$

Une résistance est un milieu peu conducteur. Les électrons peinent à s'y déplacer. Leur énergie se dissipe alors en général sous forme de chaleur. C'est ce principe utilisé pour les bouilloires électriques ou les ampoules à filaments.

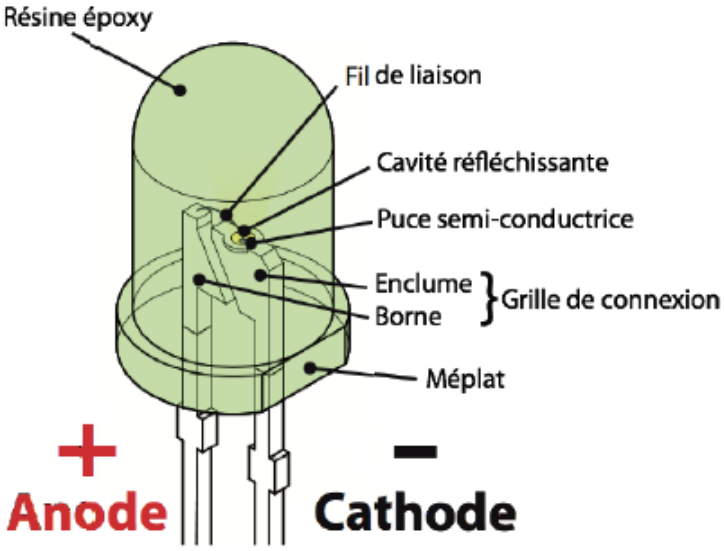
La résistance est schématisée de cette manière :

La valeur de la résistance se mesure en Ohms (Ω). La valeur d'une résistance est déterminée par ses bandes de couleurs :

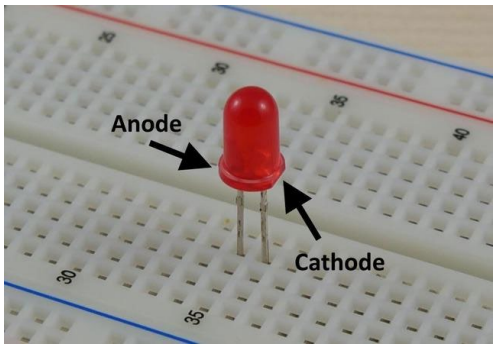


Les diodes

La diode électroluminescente, aussi appelée DEL (ou LED en anglais), a la particularité de ne laisser passer le courant électrique que dans un sens.
Le courant électrique ne peut traverser la diode que dans le sens de *l'anode* vers la *cathode*.



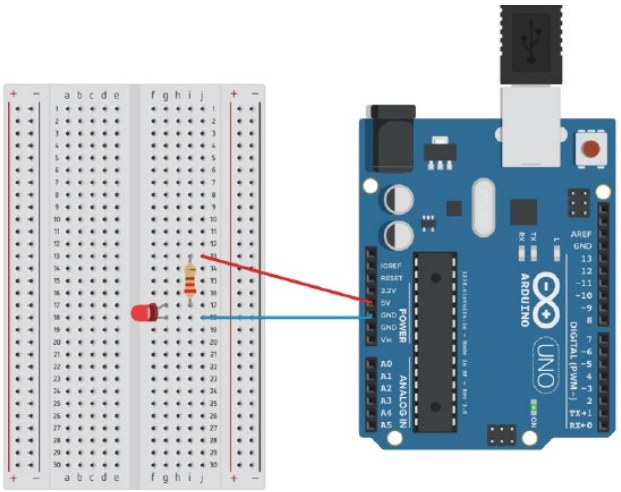
On reconnaît l'anode, car il s'agit de la broche la plus longue. Lorsque les deux broches sont de même longueur, on peut distinguer l'anode de la cathode, par un méplat du côté de cette dernière.



Le symbole de la DEL est le suivant :

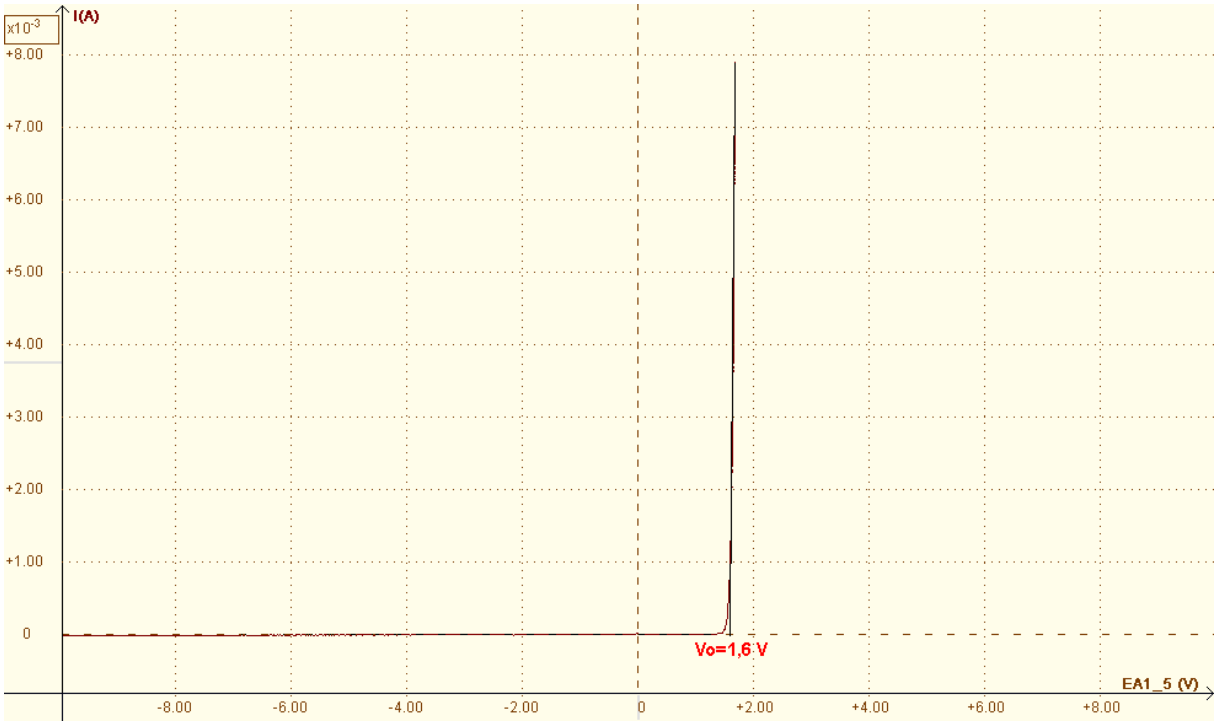
En utilisant divers matériaux semi-conducteurs, on fait varier la couleur de la lumière émise par la DEL et Il existe une grande variété de formes de DELs.

Attention : le courant produit par l'Arduino est trop important pour y brancher directement une DEL dessus.
L'utilisation d'une résistance est obligatoire, pour ne pas griller la DEL comme dans le circuit ci-dessous :



Mais si la valeur de la résistance est trop grande, la DEL ne s'allumera pas et au contraire, si la valeur de la résistance n'est pas suffisante, la DEL grillera.

Si on trace la caractéristique d'une DEL, c'est-à-dire, le graphe représentant l'intensité I traversant la DEL en fonction de la tension U à ses bornes : **I=f(U)**, on remarque qu'en dessous d'une certaine valeur de tension, le courant ne passe pas (la DEL ne s'allume pas).
On dit que la DEL est bloquante en dessous d'une tension seuil et passante au-dessus.



Caractéristique courant / tension d'une DEL rouge

Pour que la DEL s'allume, il faut donc que la tension appliquée à ses bornes soit supérieure à sa tension seuil.
La tension seuil de la DEL, U_{LED}, dépend de sa couleur. Pour connaitre sa valeur, il suffit de consulter sa fiche technique (Datasheet) donné par le constructeur de la DEL.

En général, on aura :

. DEL Blanche : U_{LED} = 3,4 à 3,8 V

. DEL Bleue : U_{LED} = 3,2 à 3,6 V

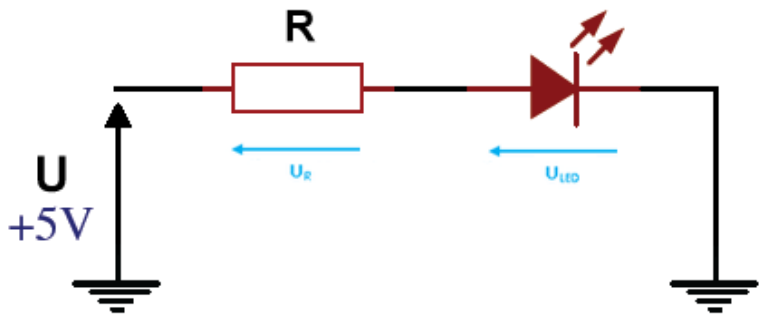
. DEL Verte : U_{LED} = 2,2 V

. DEL Rouge : U_{LED} = 1,6V à 2 V

. DEL Jaune : U_{LED} = 2,1 V

Et pour qu'une DEL fonctionne dans des conditions optimales, les constructeurs préconisent généralement un courant de **20 mA** (0,02 A) maximum.

Pour calculer la valeur de la résistance adéquate qu'il faut utiliser, on doit appliquer la loi d'Ohm au circuit suivant :



D'après la loi d'additivité des tensions dans un circuit en série, $\mathbf{U = U_R + U_{LED}}$

Donc : $\mathbf{U = Ri + U_{LED}}$

Et : $$R = \frac{U - U_{LED}}{I}$$

Par exemple :

Pour une LED rouge, U_{LED} = 1,6 V, avec une alimentation de 5 V et une Intensité de 20mA maximum, la résistance minimale est :

R = (5 - 1,6) / 0,02 = 170 Ω

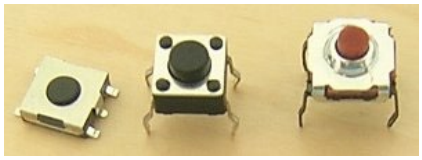
En prenant une résistance de **220 Ω**, nous sommes assurés de ne pas dépasser le courant maximal admissible par la DEL, et comme la DEL rouge est celle qui a une tension de seuil la plus basse, la résistance de 220 Ω peut être utilisée avec les autres DELs (l'intensité dans le circuit sera obligatoirement inférieure à 20 mA).

A retenir :

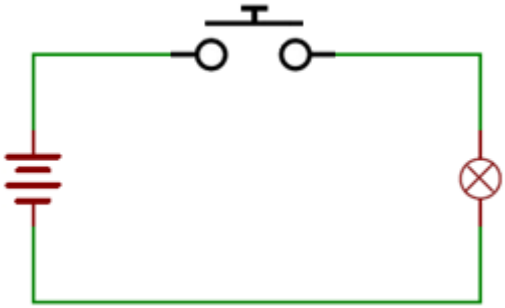
Si on utilise une DEL dans un circuit alimenté par un Arduino, il est impératif de placer une résistance, par défaut de 220 Ω, en série avec elle. Cette résistance est appelée, résistance de protection, de façon à limiter le courant qui la traverse à 20 mA maximum.

Le bouton poussoir normalement ouvert

Un bouton poussoir normalement ouvert est un dispositif mécanique doté de 4 broches et d'une lamelle métallique qui met en contact toutes les broches lorsqu'on appui sur la tête du bouton. Un ressort de rappel ramène la tête du bouton lorsqu'il est relâché.



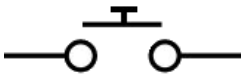
Un bouton poussoir normalement ouvert n'est jamais qu'un fil qui est connecté au circuit électrique ou non selon sa position :



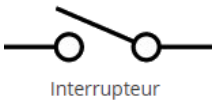
. **Relâché** : le courant ne passe pas dans le circuit électrique, le circuit est "ouvert ".
. **Appuyé** : le courant passe, on dit que le circuit est fermé.

Les mêmes effets peuvent être produits avec un interrupteur sauf que circuit reste fermé ou ouvert tant que la position de l'interrupteur n'a pas été changé.

Le bouton poussoir et l'interrupteur ne possèdent pas le même symbole pour les schémas électroniques :

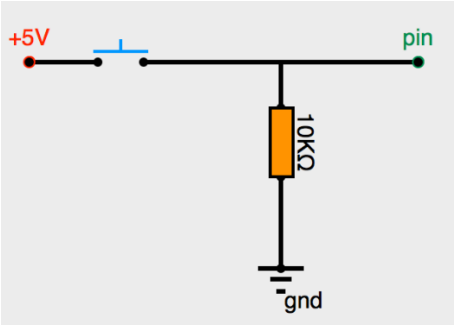


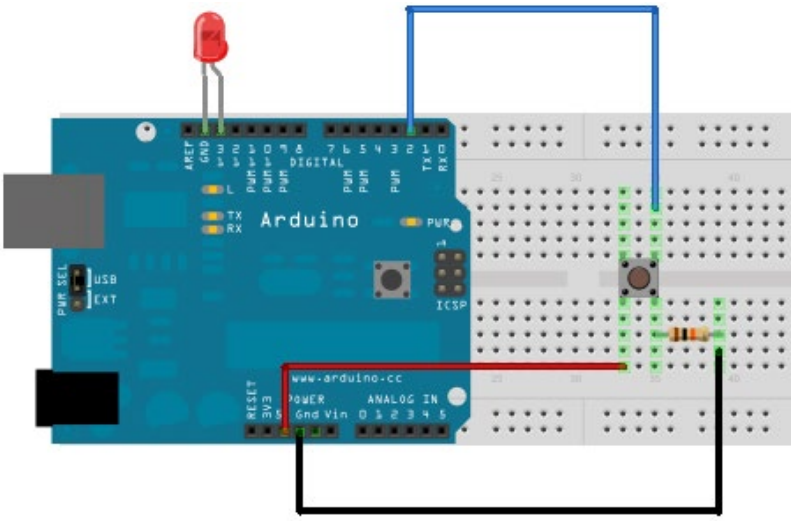
Bouton Poussoir NO



Interrupteur

Avec un Arduino, il est principalement utilisé pour envoyer une "impulsion de commande" avec ce circuit électrique :





Quand le bouton poussoir est appuyé, le potentiel de sa broche connectée à la broche 2 de l'Arduino passe à 5 V (le circuit est fermé) et quand il est relâché, celui-ci passe à 0 V (le circuit est ouvert).

On pourra alors demander à l'Arduino, d'allumer ou d'éteindre la DEL connectée sur sa broche 13 en fonction de la valeur du potentiel de la broche 2 de l'Arduino.

Avec le bouton poussoir, nous allons donner, à l'Arduino, l'ordre d'effectuer une action. D'où le terme "impulsion de commande"

<u>Attention</u> :

Il est impératif d'utiliser une résistance de 10 kΩ en série avec le bouton poussoir dans un montage "impulsion de commande".

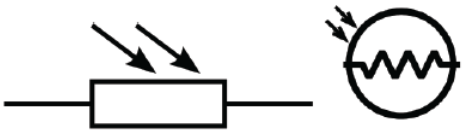
Ainsi, le courant dans le circuit est très faible (I = U/R = 0,5 mA) quand le circuit est fermé (bouton poussoir appuyé), et il n'y a pas de risque pour l'Arduino.

La photorésistance

Une photorésistance est un composant électronique dont la résistance (en Ohm Ω) varie en fonction de l'intensité lumineuse. Plus la luminosité est élevée, plus la résistance est basse. On peut donc l'utiliser comme capteur lumineux pour :

• Mesure de la lumière ambiante.
• Détecteur de lumière dans une pièce.
• Suiveur de lumière dans un robot.
• Détecteur de passage.
• …

Ses symboles électroniques sont les suivants :

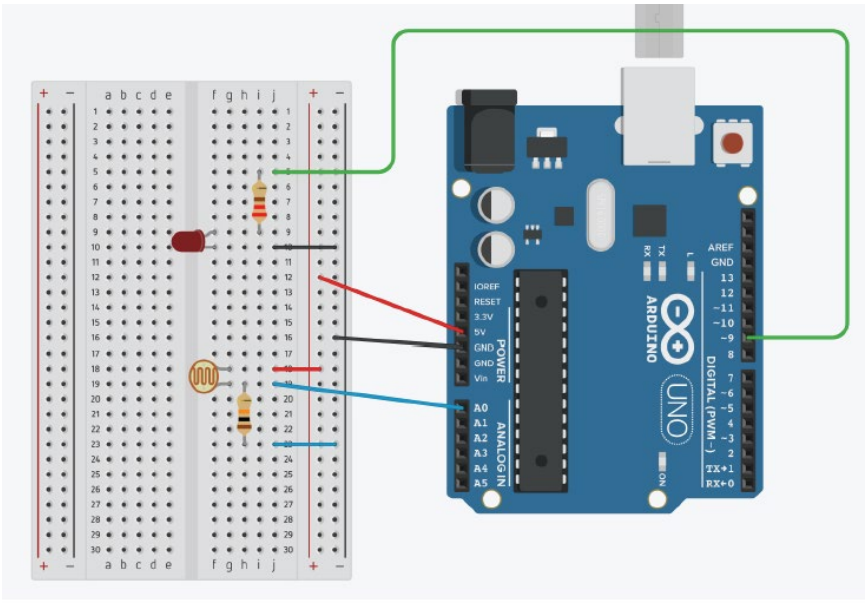


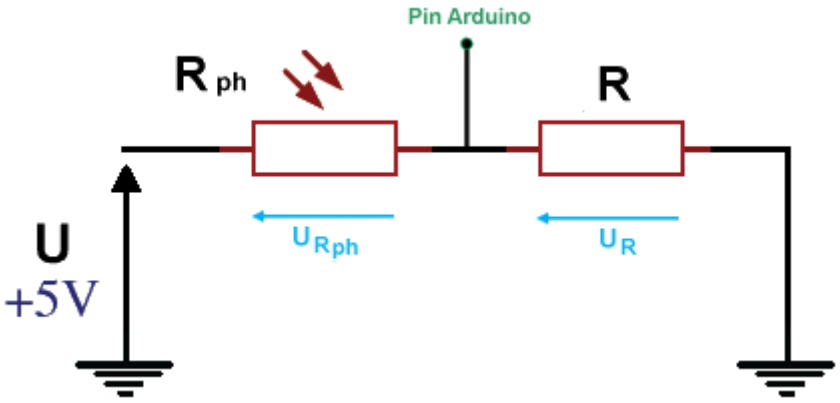
L'avantage des photorésistances est qu'elles sont très bon marché. Par contre, leur réaction à la lumière est différente pour chaque photorésistance, même quand elles ont été produites dans le même lot. On peut ainsi noter une différence de résistance de plus de 50% entre deux photorésistances du même modèle pour la même luminosité. On ne peut donc pas l'utiliser pour une mesure précise de la lumière. Par contre, elles sont idéales pour mesurer des changements simples de la luminosité.

On utilise la photorésistance dans un montage (Cf. ci-dessous), avec une résistance fixe de 10 kΩ, qu'on appelle un **diviseur de tension**.

La photorésistance est alimentée en 5V depuis l'Arduino. Le point entre les deux résistances est relié à une broche analogique de l'Arduino et on mesure la tension de cette broche par la fonction « **analogRead (broche)** ».

Tout changement de la tension mesurée est dû à la photorésistance puisque c'est la seule qui change dans ce circuit, en fonction de l'intensité lumineuse qu'elle reçoit.





D'après la loi d'additivité des tensions dans un circuit en série :

$$U = U_{Rph} + U_R = (R_{ph} + R)\, I$$

$$U_R = U - U_{Rph} = U - R_{ph}\, I$$

U_R est la tension appliquée sur l'entrée analogique de l'Arduino. Quand l'intensité lumineuse reçue par la photorésistance augmente, R_{ph} diminue, donc U_R augmente, et au contraire quand la luminosité diminue, R_{ph} augmente et U_R diminue.

On peut exprimer Ur en fonction de U :

$$U = U_{Rph} + U_R = (R_{ph} + R)\, I$$

Donc : $$I = \frac{U}{(R_{ph} + R)}$$

Et : $$U_R = R\, I = \frac{R}{(R_{ph} + R)}\, U$$

C'est la raison pour laquelle on appelle ce montage un diviseur de tension.

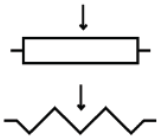
Le potentiomètre

Le potentiomètre est une résistance variable. C'est le bouton de réglage du volume que l'on retrouve sur une radio. La plupart des potentiomètres sont soit rotatifs, soit linéaires.

Potentiomètre rotatif

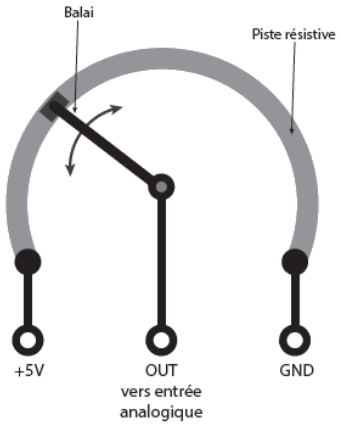
Potentiomètre linéaire

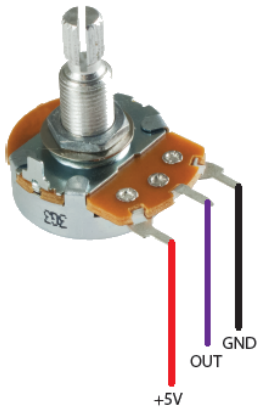
Voici les symboles électroniques (européen dessus et américain dessous) du potentiomètre :



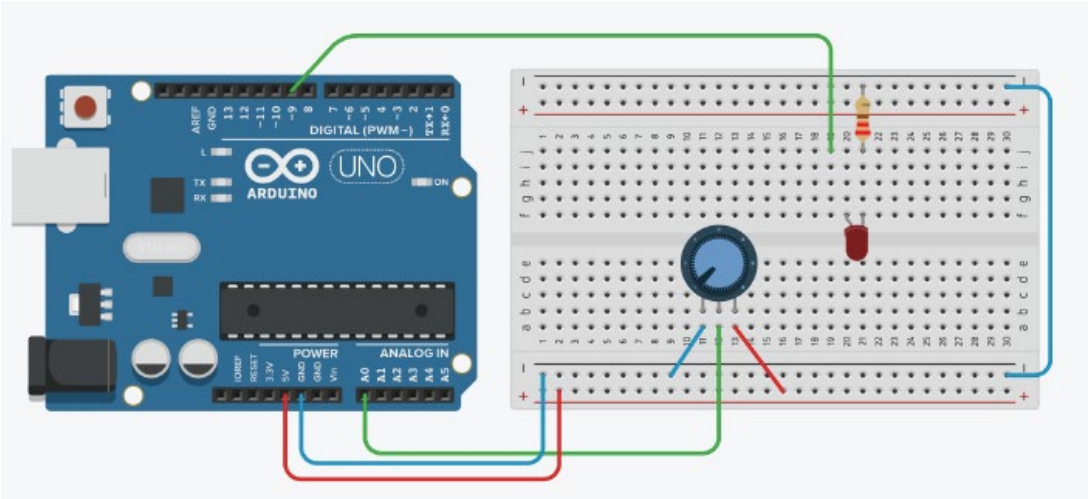
Comme toute résistance, le potentiomètre modifie la tension d'un circuit. On va donc l'utiliser principalement comme entrée (input) sur une broche analogique (A0 à A5) de l'Arduino.

Les potentiomètres ont en général trois broches.

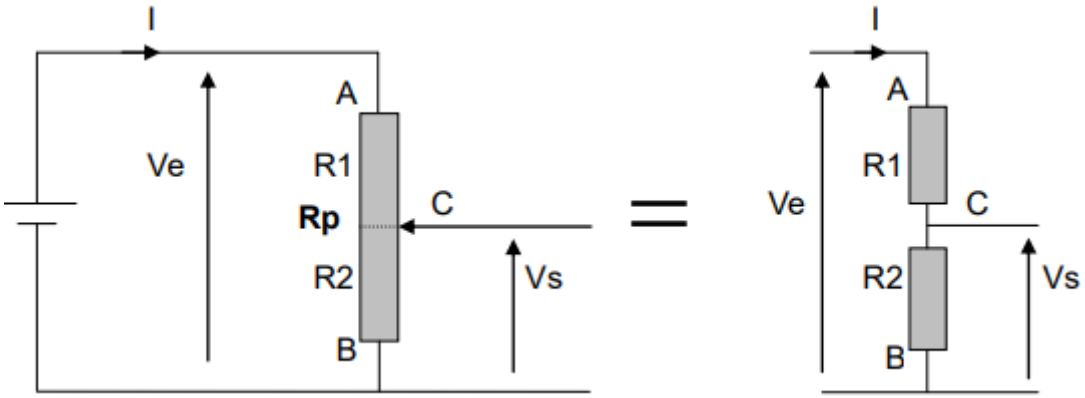




Les broches extérieures se connectent sur l'alimentation +5V et sur la terre, alors que la broche centrale envoie le signal sur la broche d'entrée analogique de l'Arduino, comme sur le circuit ci-dessous :



Dans ce montage, le potentiomètre de résistance totale Rp est utilisé en pont diviseur de tension :



On a : **Ve = (R1 + R2) I**

$$\mathbf{I = \frac{Ve}{(R_1 + R_2)}}$$

Et : $\mathbf{Vs = R_2\, I = \frac{R_2}{(R_1 + R_2)}\, Ve = \frac{R_2}{Rp}\, Ve}$ **(car $R_p = R_1 + R_2$)**

On peut remplacer le rapport $\mathbf{R_2 / Rp}$ par la position du curseur comprise entre 0 (position B) et 1 (position A).

Dans ce cas, la relation devient :

Vs = α Ve (avec α la position du curseur : $0 \leq \alpha \leq 1$)

Vs, qui est la tension appliquée sur une entrée analogique de l'Arduino, varie donc entre 0 et +5V en fonction de la position du curseur du potentiomètre.

De façon à limiter le courant dans le circuit à 0,5 mA, on utilise généralement des potentiomètres de **10 kΩ**.

La DEL RVB

Une DEL RVB (RVB pour Rouge-Vert-Bleu), n'est rien d'autre qu'un ensemble de 3 DELs, une rouge une verte et une bleue rassemblées dans un seul et même boitier.

Une DEL RVB a 4 broches : une commune à l'ensemble des DELs et une pour chaque couleur de la DEL. La broche commune pourra, selon les modèles, être le + (anode commune) ou le - (cathode commune).

La DEL ci-dessous a pour broche commune la cathode :

2.5. Les projets (étude de circuits) - Les activités

Les premiers circuits à étudier et les premières activités associées permettent de se familiariser avec l'Arduino, de comprendre le principe de fonctionnement des entrées et sorties numériques ou analogiques.

Ensuite, c'est l'exploitation des données des principaux capteurs utilisés avec un Arduino qui sera abordée.

Et enfin, on étudiera la gestion des moteurs et servomoteurs.

Pour tous les circuits, quel que soit le mode de fonctionnement d'**ARDUINO LAB** choisi ("**Contrôle de l'Arduino**" ou "**Simulation**"), il faut cliquer sur le connecteur USB pour :

- soit établir la connexion avec l'Arduino et commencer à donner des ordres ou recevoir des données, dans le cas où le mode fonctionnement est le contrôle de l'Arduino,

- soit lancer la simulation, dans l'autre cas.

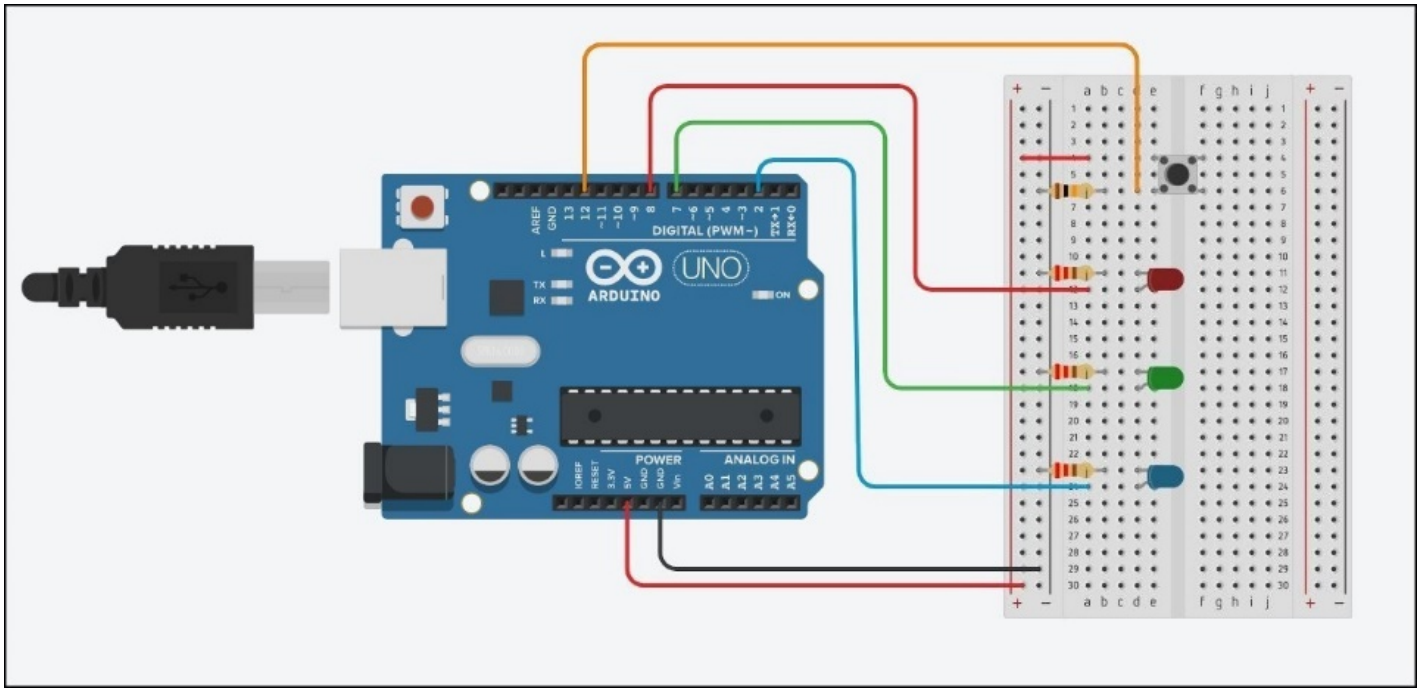
Recommandations :

- Pour le mode de fonctionnement "**Contrôle de l'Arduino**", il faut penser à téléverser le protocole de communication adéquat entre **ARDUINO LAB** et la platine Arduino ou sélectionner le mode sélection automatique du protocole dans le menu "**Paramètres**"

- Il est conseillé de cliquer sur la prise USB avant de quitter l'étude du projet ou de fermer **ARDUINO LAB,** afin de fermer le port "**COM**" sur lequel l'Arduino est connecté.

2.5.1 Premiers pas : DEL & Bouton poussoir

L'objectif de l'étude de ce premier circuit est de comprendre le principe de fonctionnement des entrées et sorties numériques de l'Arduino à travers 4 activités.



- Liste des composants :

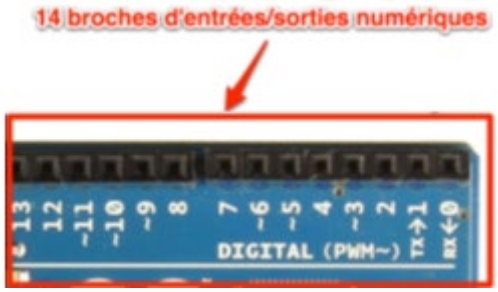
. 1 DEL rouge
. 1 DEL verte
. 1 DEL bleue
. 3 résistances de 220 Ω
. 1 bouton poussoir
. 1 plaque d'essai
. Fils de connexion

- Protocole de communication (Mode "Contrôle de l'Arduino") :

. Firmata standard

Rappel :

. Les entrées et sorties numériques



Il y a 14 entrées/sorties numériques notées de 0 à 13 sur l'Arduino Uno. Elles sont situées sur la grande rangée du haut de la carte. Ces connecteurs (ou broches) sont numériques car le signal sur ces broches ne connaît que deux états (niveau logique) : haut ou bas (1 ou 0). Électriquement, cela se traduit, respectivement, par une tension de 5 V ou 0 V.

Ces broches sont configurées en entrées ou sorties numériques par programmation :

. Configurées en sortie, elles ne peuvent délivrer que des niveaux logiques bas (0 V) et des niveaux logiques haut (5 V).

. Configurées en entrée, elles ne peuvent recevoir que des niveaux logiques bas (0 V) et niveaux logiques haut (5 V).

Attention :

. Les tensions appliquées sur les broches d'entrées/sorties numériques doivent être comprise entre 0 et 5 V. Au-dehors de ces limites, le microcontrôleur sera endommagé.

. Pour une broche configurée en entrée, toute tension inférieure à **0,3 x Vcc**, Vcc étant égale à 5V, soit **1,5 V**, sera comprise comme un niveau logique bas (0 V) et toute tension supérieure à **0,6 x Vcc**, soit **3 V**, sera comprise comme un niveau logique haut (5 V).

Entre les deux, c'est incertain. L'Arduino renverra de toutes façons un 0 ou un 1 mais de manière plus ou moins aléatoire.

. Pour une broche configurée en sortie, il est préférable de limiter l'intensité du courant dans le circuit électrique à **20 mA** et absolument nécessaire de ne pas dépasser **40 mA** sous peine de destruction de la sortie. On veillera aussi à ne pas dépasser une intensité totale de **200 mA** dans les circuits électriques reliés à ces broches.

- Activité 1 : Faire clignoter une DEL

Comme première activité, nous allons faire clignoter une DEL (rouge, verte, ou bleue), préalablement choisie, connectée sur une des broches 8, 7, ou 2, comme sur le schéma de câblage ci-dessus.
Cette activité a pour but l'apprentissage de l'utilisation des sorties digitales de l'Arduino qui ne peuvent prendre que 2 valeurs : **0 (niveau bas)** ou **1 (niveau haut)**, soit électriquement : **0 V ou +5 V**.
Donc, pour allumer la DEL, la broche de l'Arduino sur laquelle celle-ci est connectée, doit être au niveau haut (**+5V**) et pour l'éteindre, elle doit être au niveau bas (**0 V**).

Pour réaliser cette activité, on va demander à l'Arduino d'allumer une des 3 DELs (**donc d'appliquer un niveau haut sur la broche de la DEL**) pendant ½ seconde, puis de l'éteindre (**donc d'appliquer un niveau bas sur la broche de la DEL**) pendant une ½ seconde, puis à nouveau de l'allumer pendant ½ seconde et cela indéfiniment. De cette façon, on verra la DEL choisie clignoter.

Après avoir cliqué sur le connecteur USB, le choix de la DEL est fait par l'intermédiaire de ce menu :

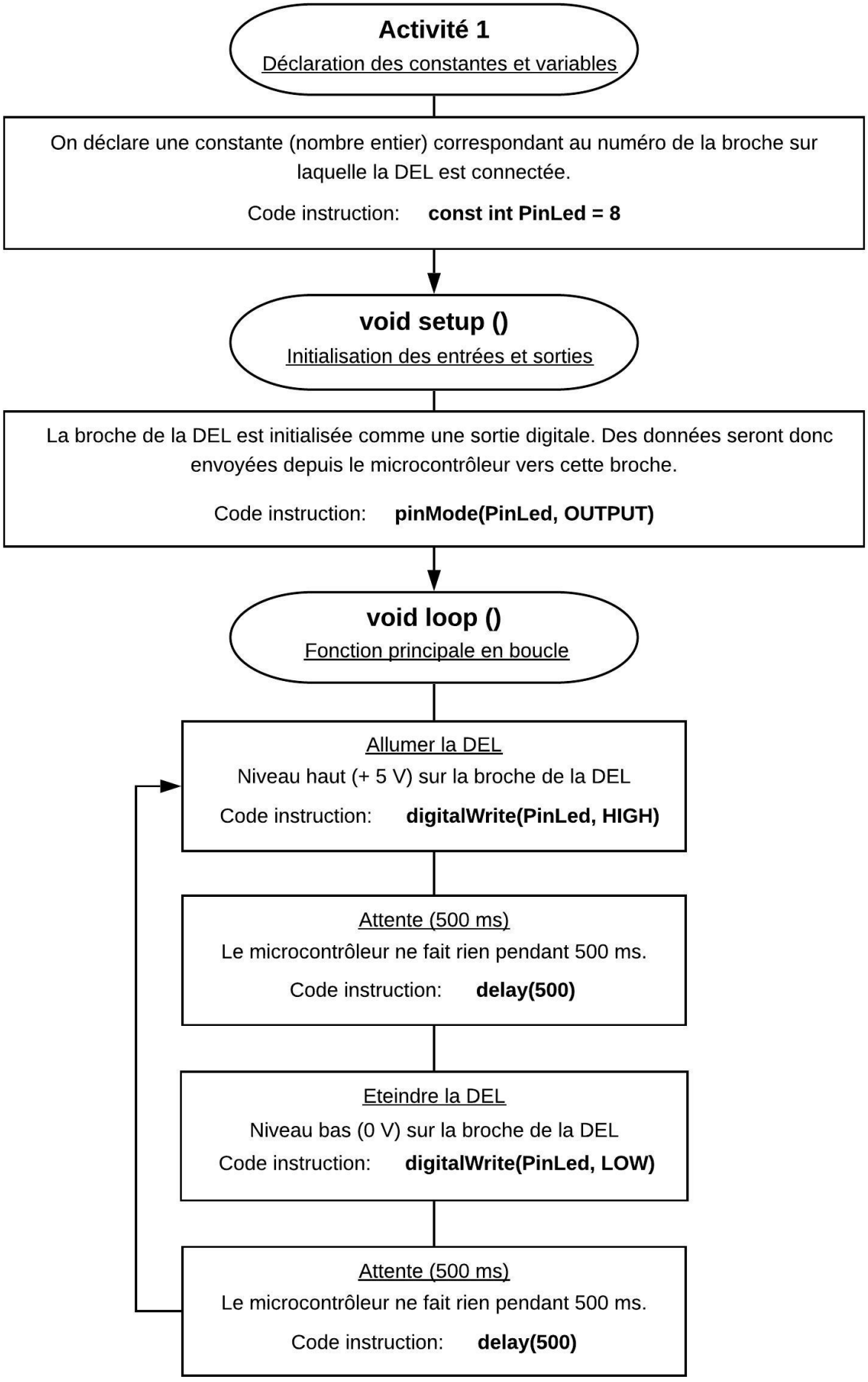
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DEL sur l'écran clignotent.

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

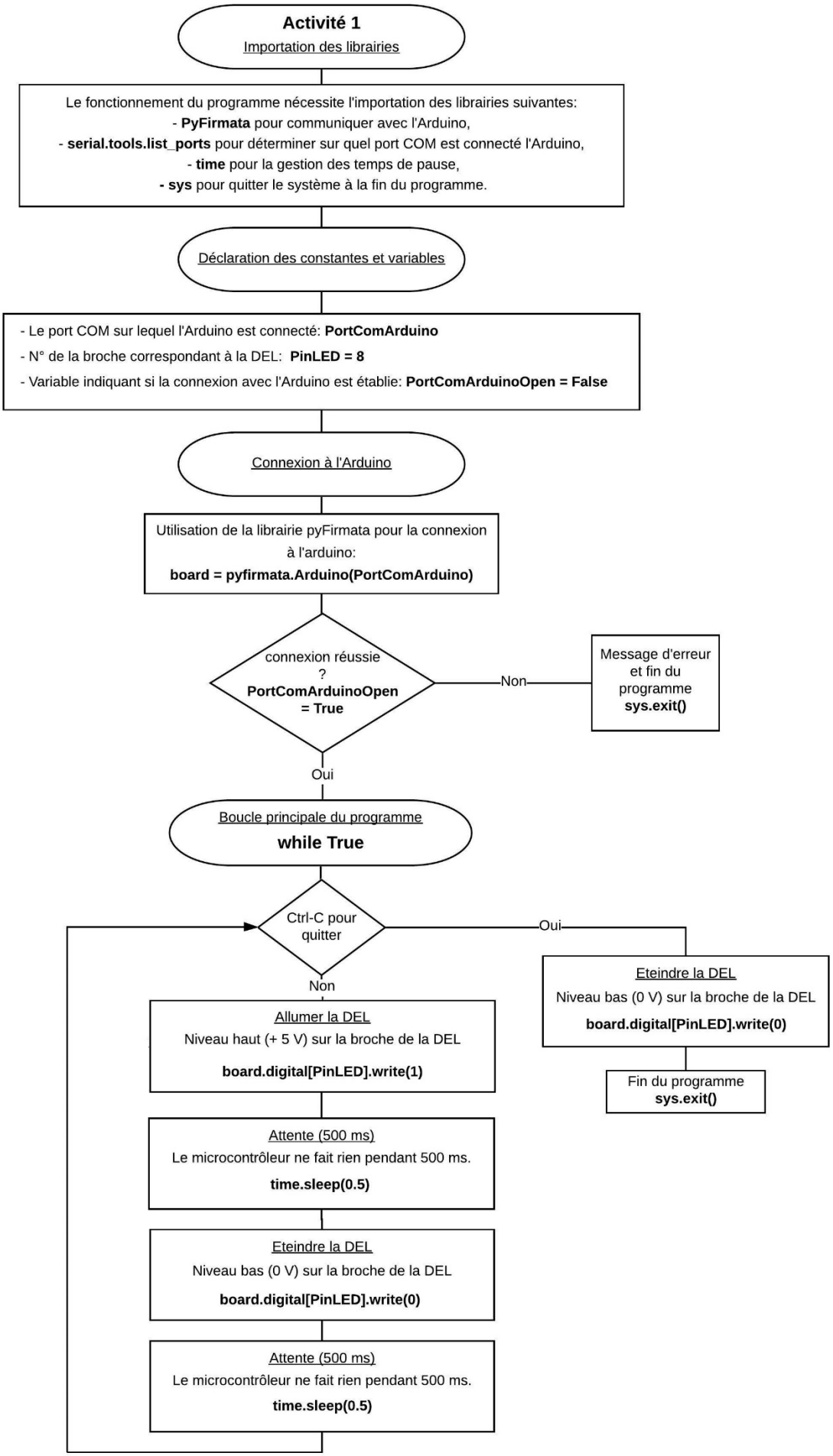
Le code pourra être modifié pour voir l'influence des variables (durée d'allumage, d'extinction, numéro de la broche de la DEL).

Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 1 en langage Arduino IDE :



* Algorithme de programmation de l'activité 1 en Python :



- Activité 2 : Allumer une DEL avec un bouton-poussoir

Dans cette activité, la DEL, préalablement choisie, s'allume en appuyant sur le bouton poussoir et s'éteint si on le relâche. L'objectif est de se familiariser avec les entrées numériques de l'Arduino.

En effet, en appuyant sur le bouton poussoir, une tension de + 5V est appliquée sur la broche sur laquelle il est connecté. La broche est alors à un niveau haut. Si on relâche le bouton poussoir, le circuit électrique est ouvert, la tension sur la broche du bouton poussoir est alors de 0 V et passe à un niveau bas.

Si on demande à l'Arduino d'interroger l'état logique (**niveau haut ou bas**) de la broche du bouton poussoir qui a été déclaré comme une entrée numérique, on peut savoir si celui-ci est appuyé ou pas et donc lui donner l'ordre d'allumer ou d'éteindre la DEL.

Après avoir cliqué sur le connecteur USB, le choix de la DEL est fait par l'intermédiaire de ce menu :

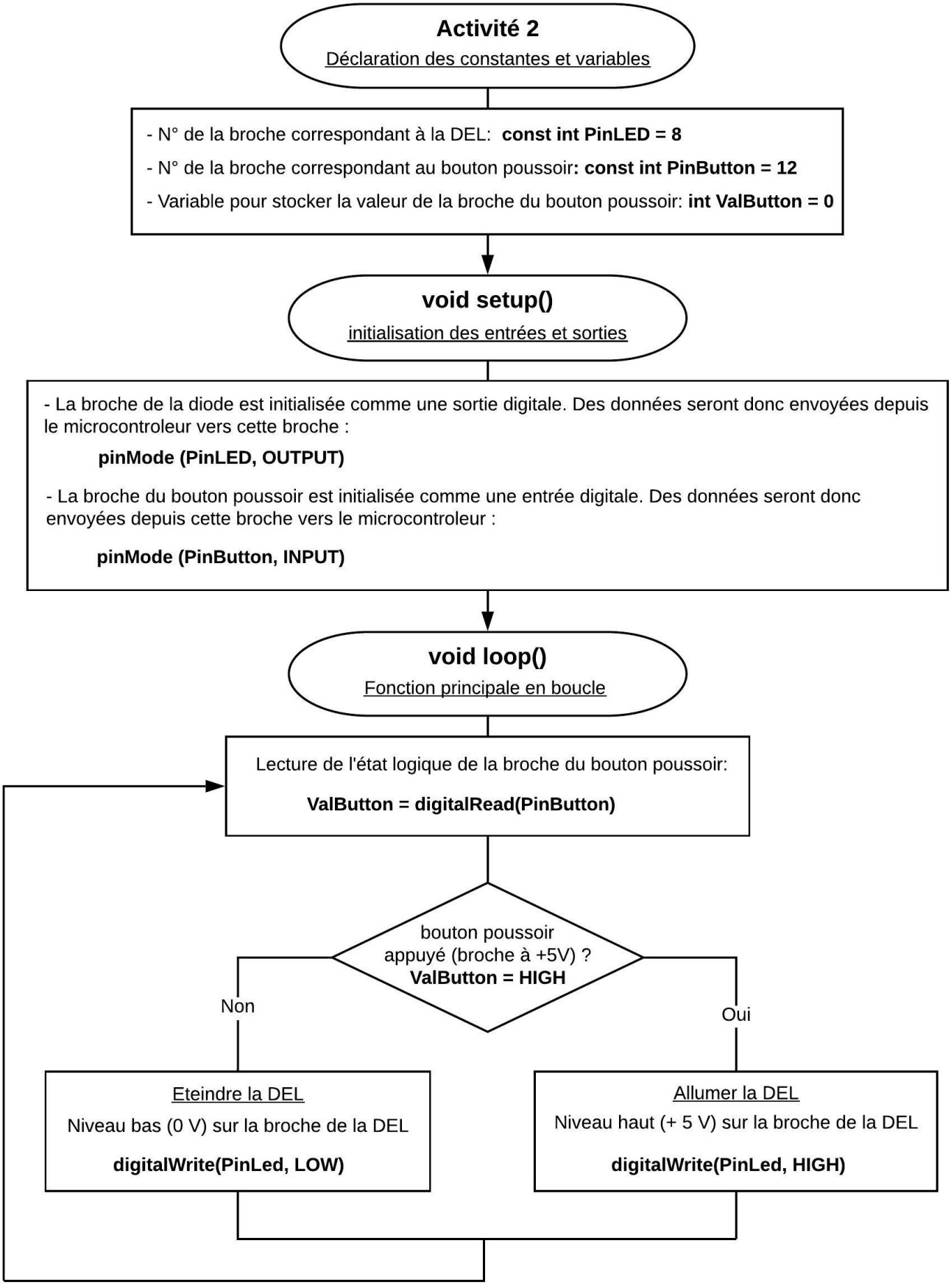
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DEL sur l'écran s'allument en appuyant sur le bouton poussoir du circuit réel ou sur celui du circuit affiché sur l'écran.

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

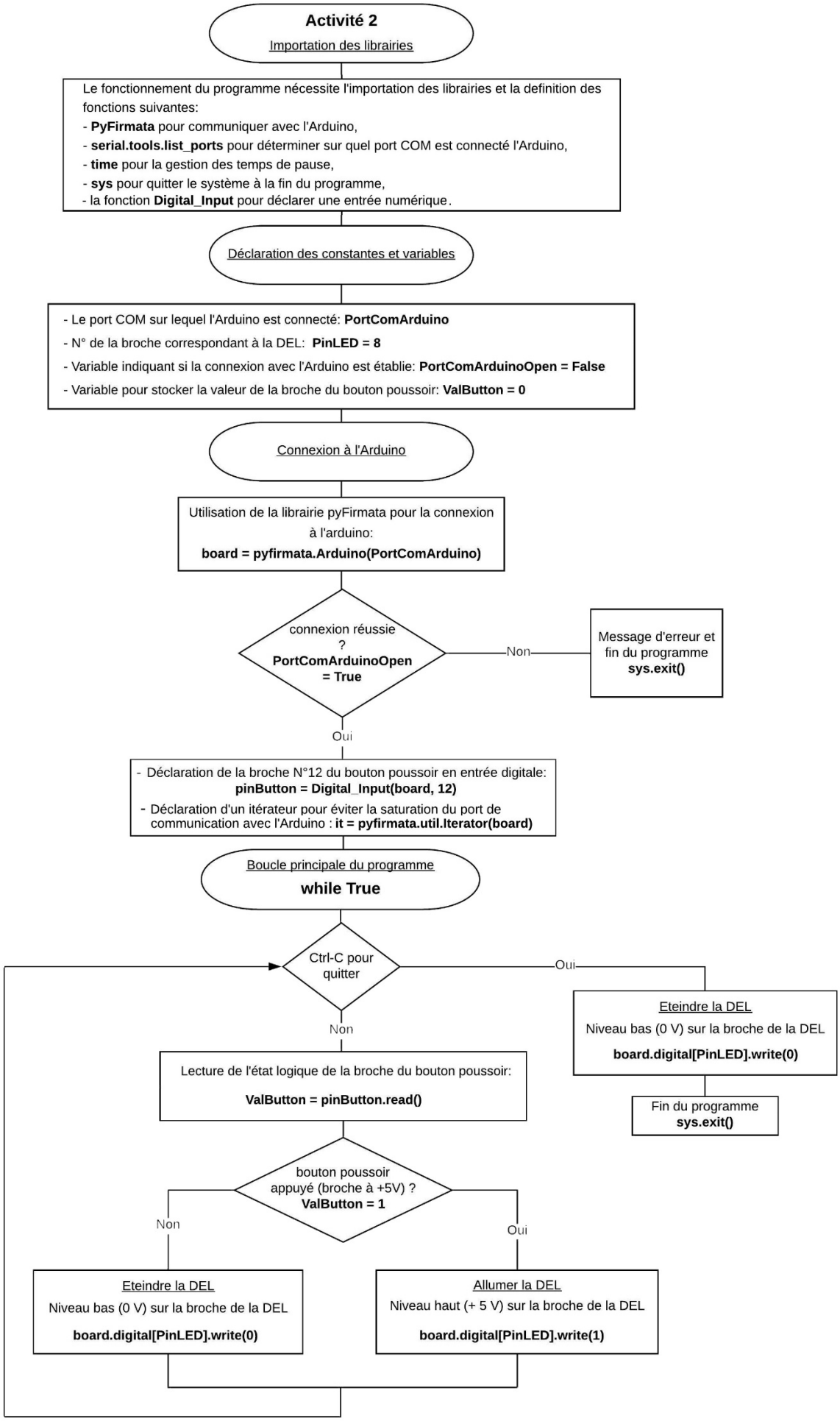
Le code pourra être modifié pour voir l'influence des variables (numéro de la broche de la DEL).

Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 2 en langage Arduino IDE :



* Algorithme de programmation de l'activité 2 en Python :



- Activité 3 : Allumer ou éteindre une DEL avec un bouton-poussoir

Dans cette activité, quand les DELs sont éteintes, si on appuie sur le bouton poussoir, une DEL préalablement choisie s'allume, mais contrairement à l'activité précédente la DEL ne s'éteint pas quand on relâche le bouton poussoir. En effet, si une des DELs est allumée, elle s'éteint en appuyant à nouveau sur le bouton poussoir. Cette fois, le principe de fonctionnement du bouton poussoir est comme celui d'un interrupteur.

Pour réaliser cette activité, on va demander à l'Arduino d'interroger l'état logique (**niveau haut ou bas**) de la broche du bouton poussoir qui a été déclaré comme une entrée numérique. A l'aide de variables permettant de stocker les valeurs (actuelle et précédente) de cet état, l'Arduino pourra savoir quelle action effectuer (allumer ou éteindre la DEL) après l'appui sur le bouton poussoir.

Après avoir cliqué sur le connecteur USB, le choix de la DEL est fait par l'intermédiaire de ce menu :

Une modification dans le choix de la DEL quand une DEL est déjà allumée, entraine une réinitialisation du programme. Il faut appuyer de nouveau sur le bouton poussoir pour allumer la DEL choisie.

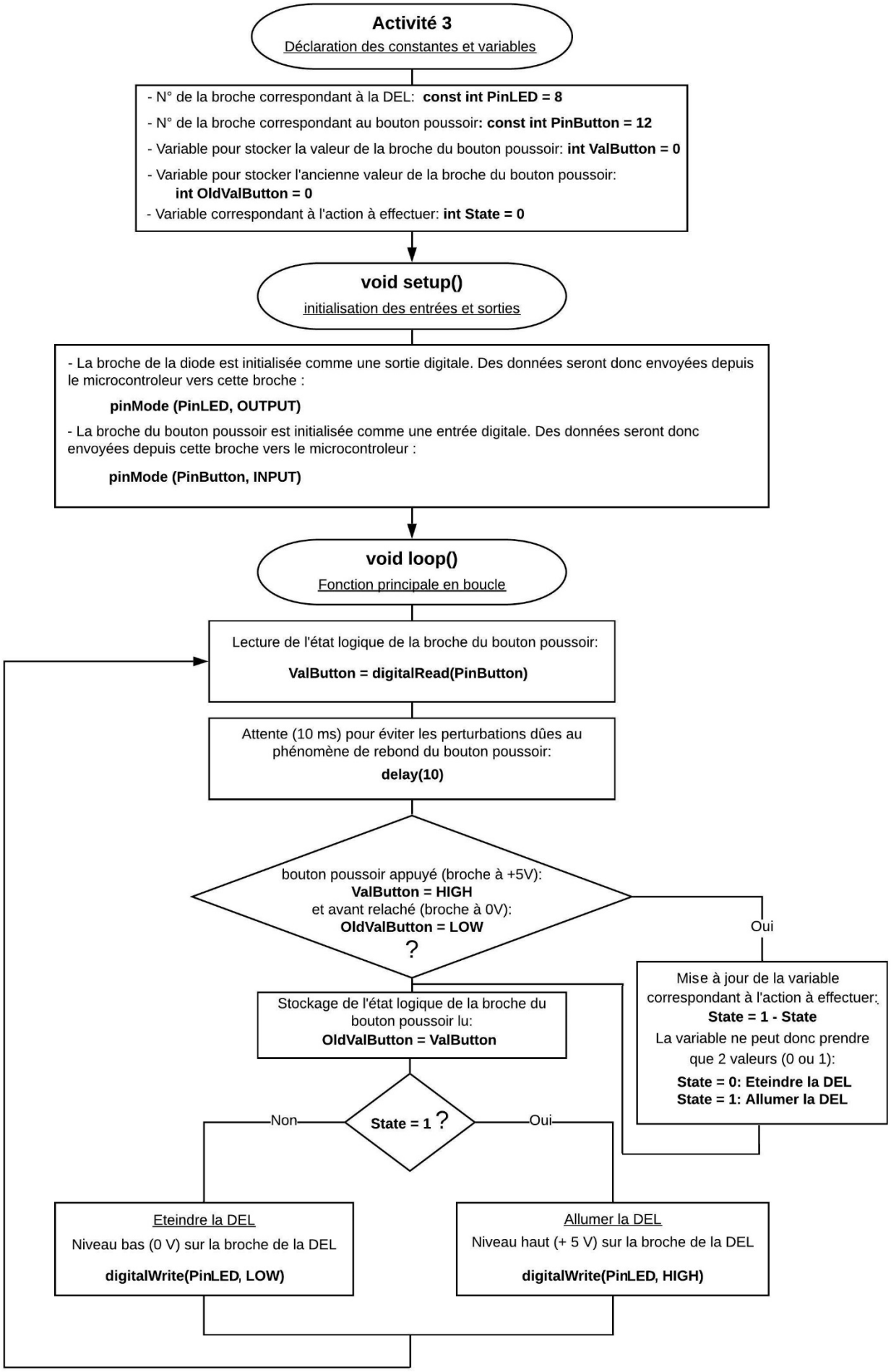
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DEL sur l'écran s'allument ou s'éteignent en appuyant sur le bouton poussoir du circuit réel ou sur celui du circuit affiché sur l'écran.

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

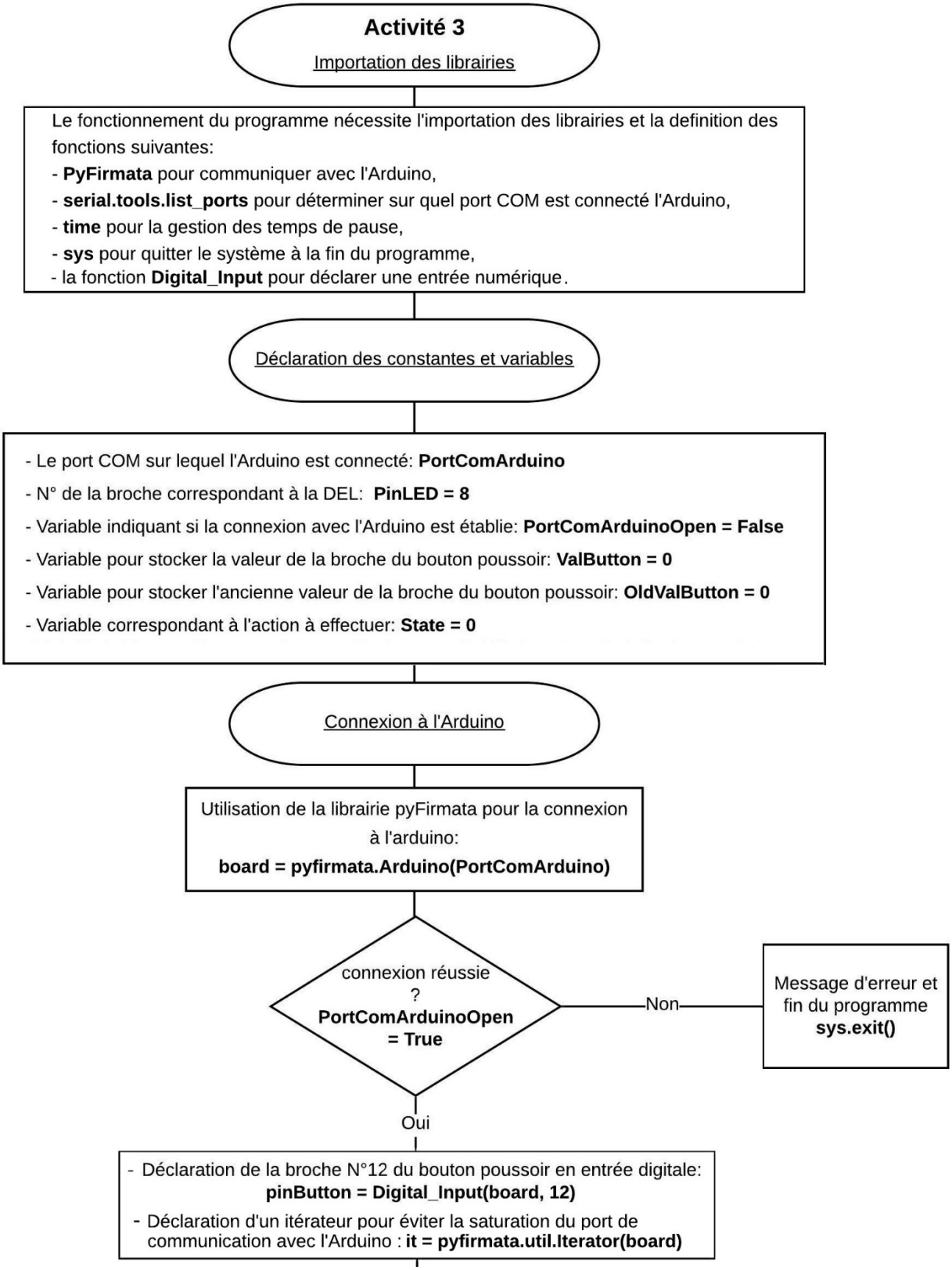
Le code pourra être modifié pour voir l'influence des variables (numéro de la broche de la DEL).

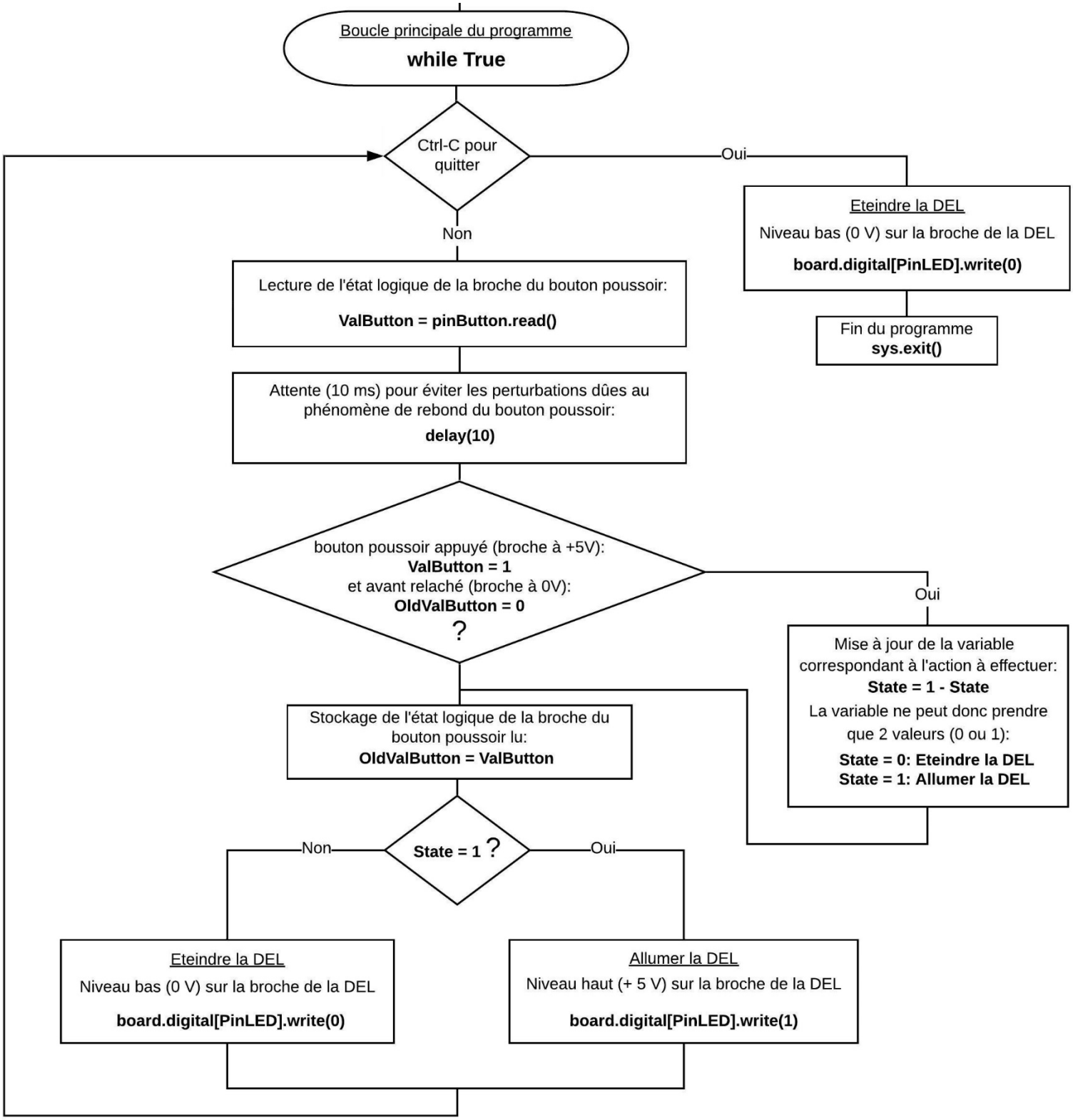
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 3 en langage Arduino IDE :



* Algorithme de programmation de l'activité 3 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre la DEL
Niveau bas (0 V) sur la broche de la DEL
board.digital[PinLED].write(0)
Fin du programme
sys.exit()
Lecture de l'état logique de la broche du bouton poussoir:
ValButton = pinButton.read()
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
Mise à jour de la variable correspondant à l'action à effectuer:
State = 1 - State
La variable ne peut donc prendre que 2 valeurs (0 ou 1):
State = 0: Eteindre la DEL
State = 1: Allumer la DEL
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Non
Oui
Eteindre la DEL
Niveau bas (0 V) sur la broche de la DEL
board.digital[PinLED].write(0)
Allumer la DEL
Niveau haut (+ 5 V) sur la broche de la DEL
board.digital[PinLED].write(1)

- Activité 4 : Allumer en alternance ou éteindre 3 DELs avec un bouton-poussoir

Dans cette activité, l'allumage en alternance des DELs est géré par le bouton poussoir. Un premier appui sur le bouton allume la diode rouge, un deuxième appui allume la diode verte, un troisième appui allume la diode bleue et ainsi de suite…
Un appui prolongé sur le bouton éteint la DEL allumée.

Comme pour l'activité précédente, c'est à l'aide des variables permettant de stocker les valeurs (actuelle et précédente) de l'état logique de la broche du bouton poussoir, mais aussi d'une variable pour compter le nombre d'appui sur le bouton et de variables pour mesurer la durée d'appui, que l'Arduino pourra allumer ou éteindre les DELs.

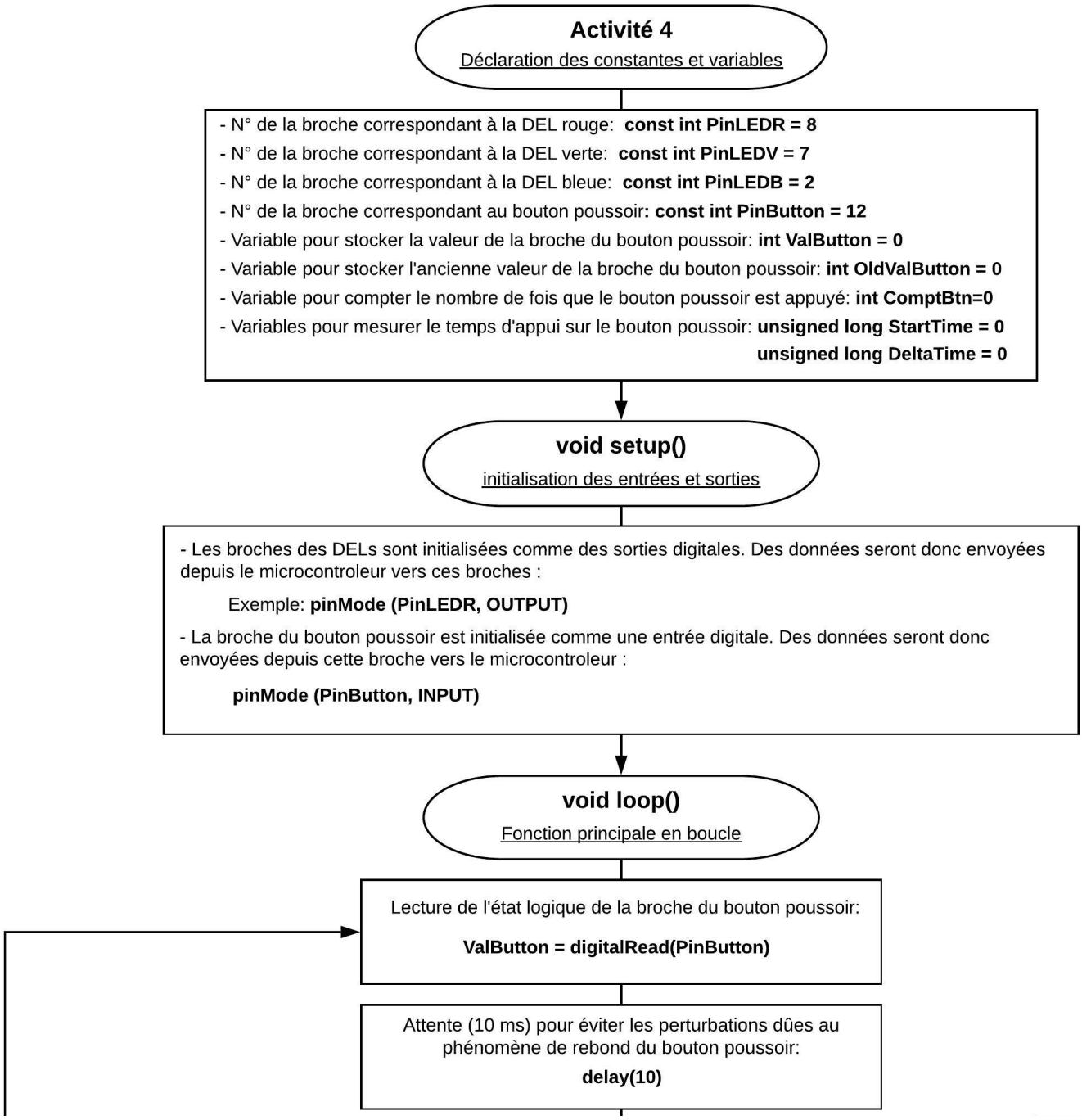
Si le mode de fonctionnement est le "contrôle de l'Arduino", les DELs du circuit réel et les DELs sur l'écran s'allument ou s'éteignent en appuyant sur le bouton poussoir du circuit réel ou sur celui du circuit affiché sur l'écran.

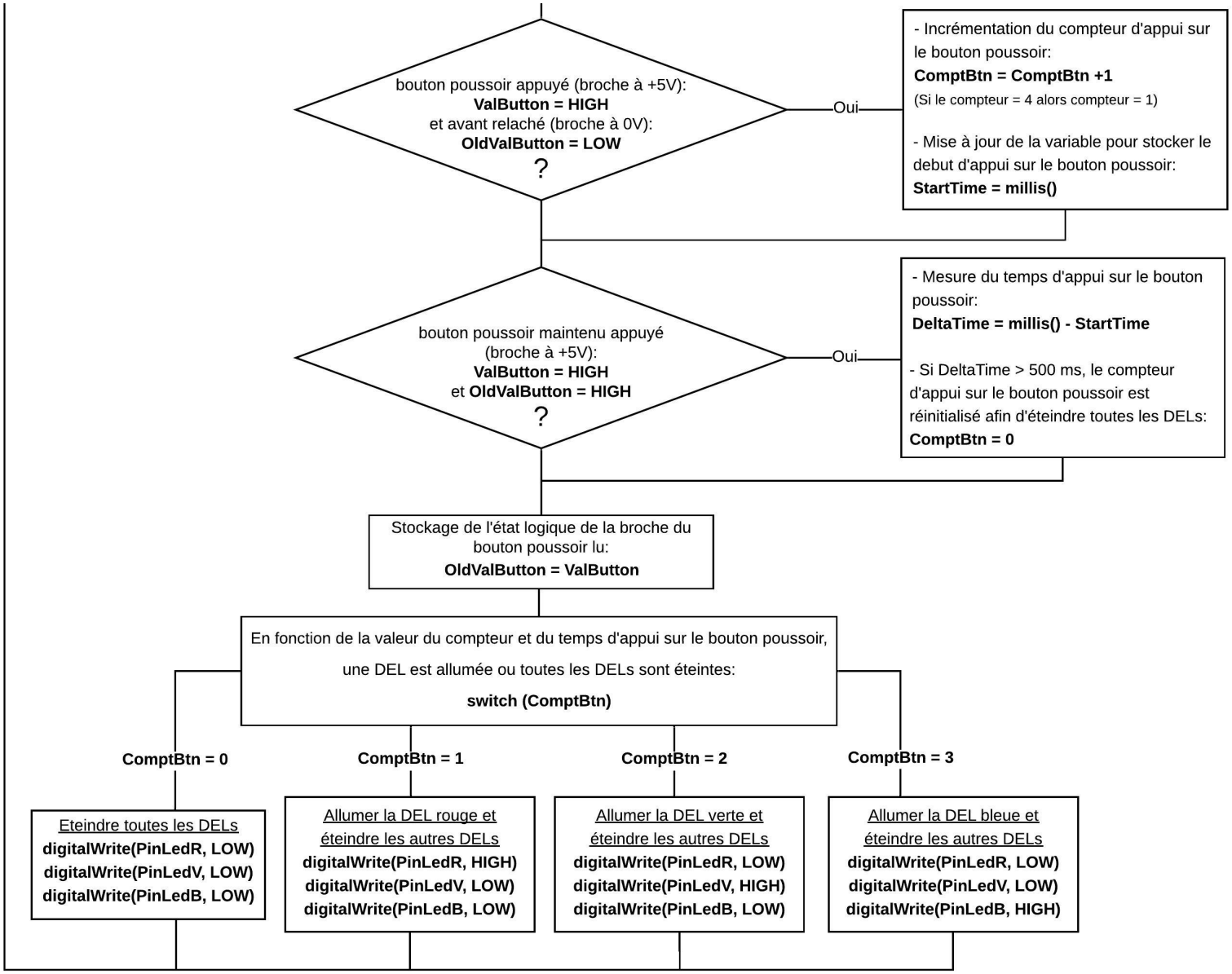
A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

Le code pourra être modifié pour voir l'influence des variables (durée d'appui pour éteindre les DELS).

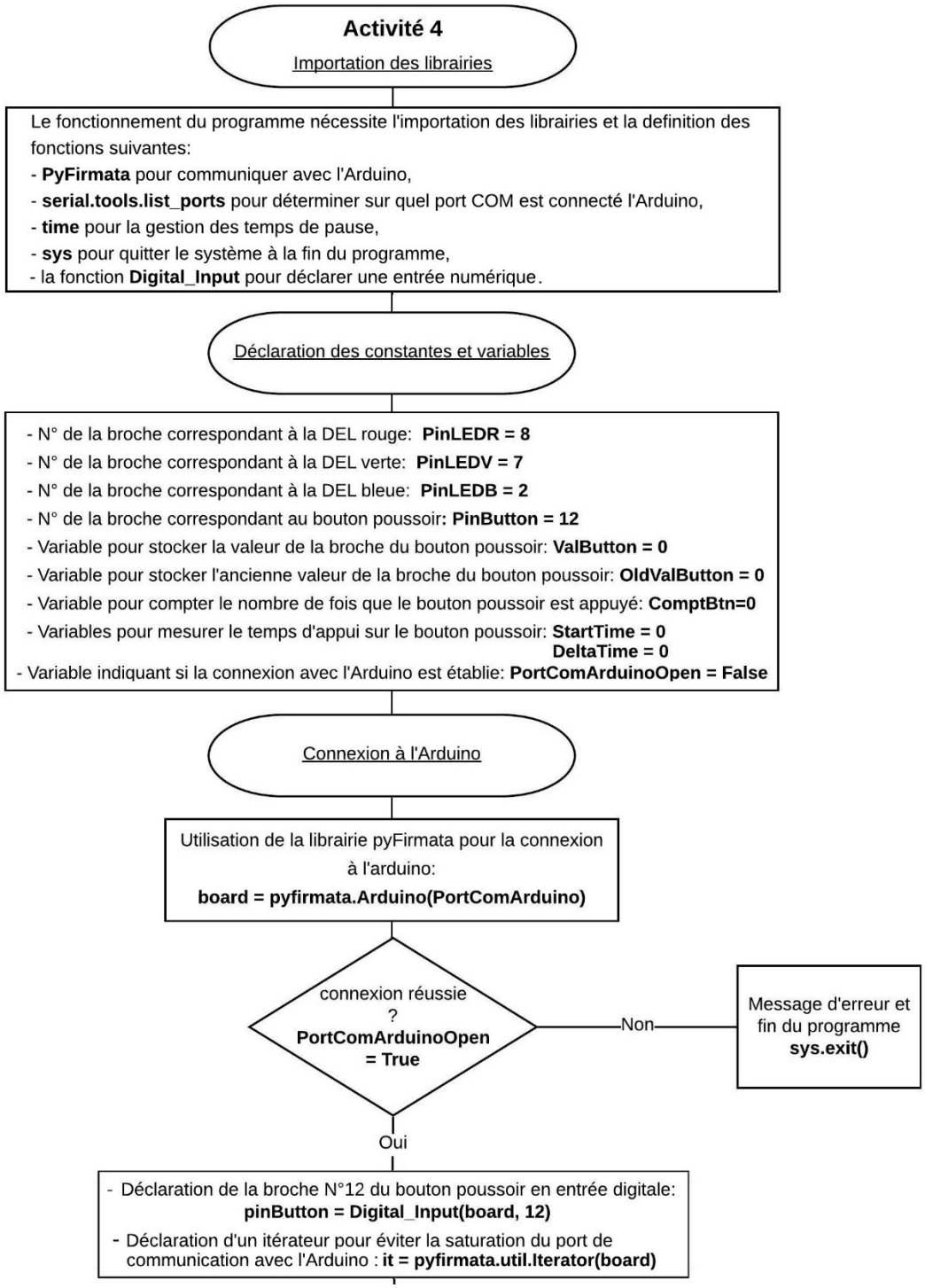
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

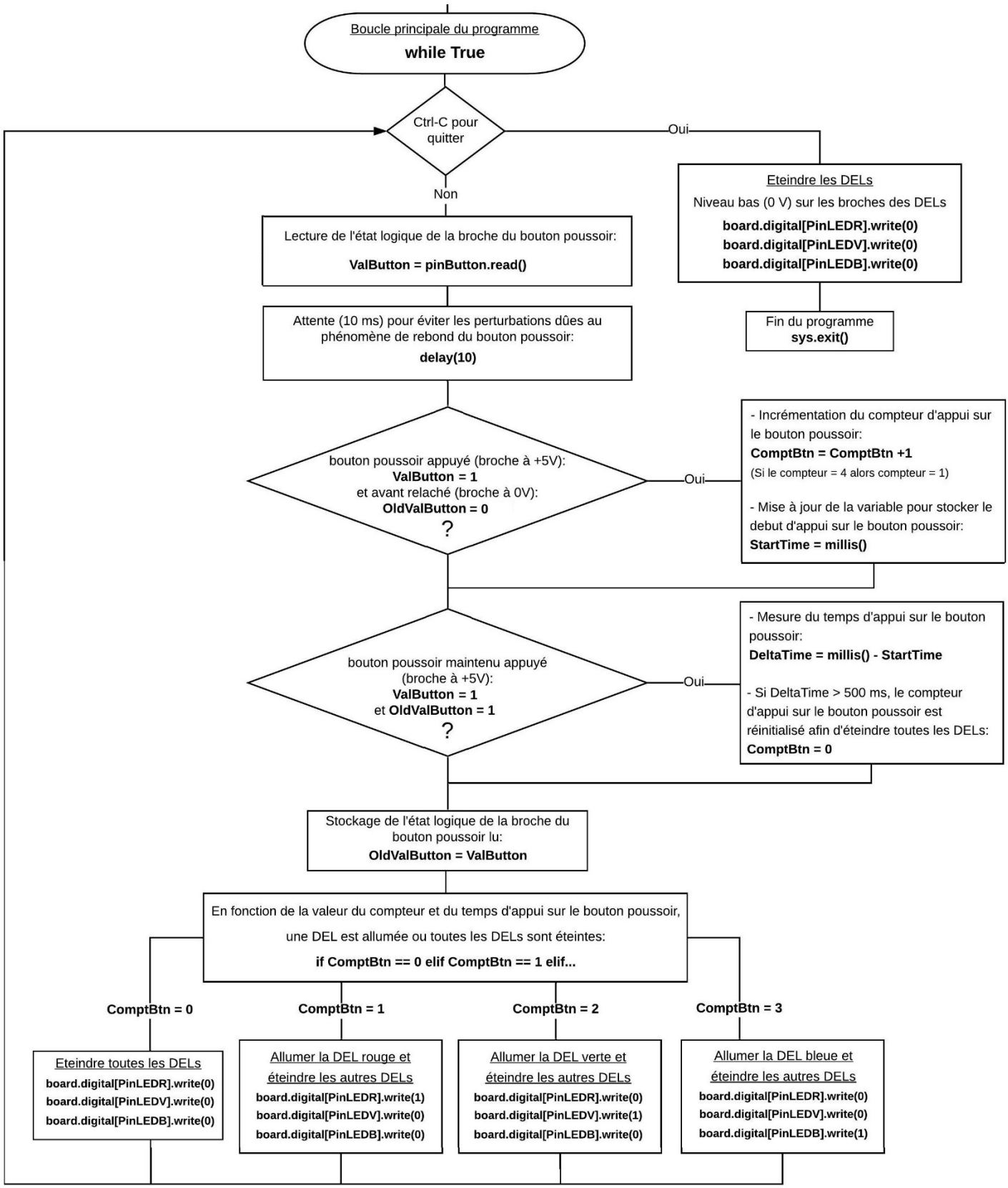
* Algorithme de programmation de l'activité 4 en langage Arduino IDE :




bouton poussoir appuyé (broche à +5V):
ValButton = HIGH
et avant relaché (broche à 0V):
OldValButton = LOW
?
Oui
- Incrémentation du compteur d'appui sur le bouton poussoir:
ComptBtn = ComptBtn +1
(Si le compteur = 4 alors compteur = 1)
- Mise à jour de la variable pour stocker le debut d'appui sur le bouton poussoir:
StartTime = millis()
bouton poussoir maintenu appuyé (broche à +5V):
ValButton = HIGH
et OldValButton = HIGH
?
Oui
- Mesure du temps d'appui sur le bouton poussoir:
DeltaTime = millis() - StartTime
- Si DeltaTime > 500 ms, le compteur d'appui sur le bouton poussoir est réinitialisé afin d'éteindre toutes les DELs:
ComptBtn = 0
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
En fonction de la valeur du compteur et du temps d'appui sur le bouton poussoir, une DEL est allumée ou toutes les DELs sont éteintes:
switch (ComptBtn)
ComptBtn = 0
ComptBtn = 1
ComptBtn = 2
ComptBtn = 3
Eteindre toutes les DELs
digitalWrite(PinLedR, LOW)
digitalWrite(PinLedV, LOW)
digitalWrite(PinLedB, LOW)
Allumer la DEL rouge et éteindre les autres DELs
digitalWrite(PinLedR, HIGH)
digitalWrite(PinLedV, LOW)
digitalWrite(PinLedB, LOW)
Allumer la DEL verte et éteindre les autres DELs
digitalWrite(PinLedR, LOW)
digitalWrite(PinLedV, HIGH)
digitalWrite(PinLedB, LOW)
Allumer la DEL bleue et éteindre les autres DELs
digitalWrite(PinLedR, LOW)
digitalWrite(PinLedV, LOW)
digitalWrite(PinLedB, HIGH)

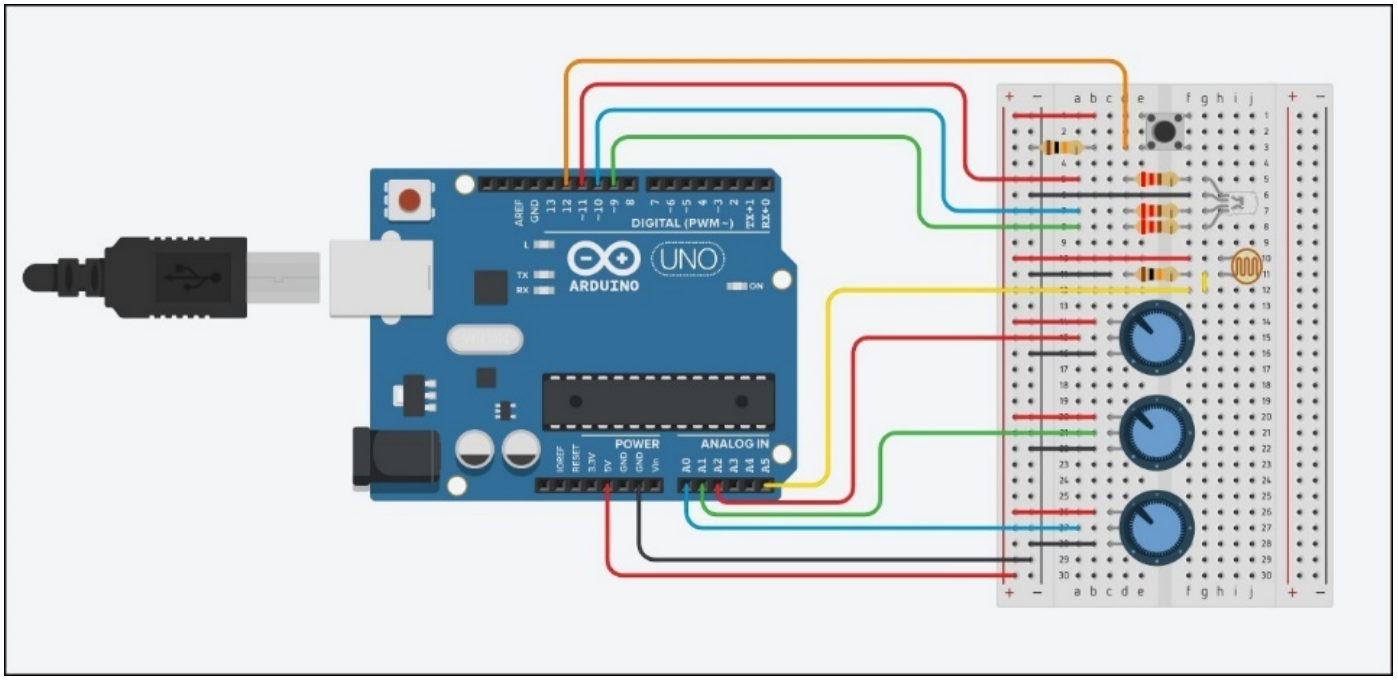
* Algorithme de programmation de l'activité 4 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre les DELs
Niveau bas (0 V) sur les broches des DELs
board.digital[PinLEDR].write(0)
board.digital[PinLEDV].write(0)
board.digital[PinLEDB].write(0)
Fin du programme
sys.exit()
Lecture de l'état logique de la broche du bouton poussoir:
ValButton = pinButton.read()
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Incrémentation du compteur d'appui sur le bouton poussoir:
ComptBtn = ComptBtn +1
(Si le compteur = 4 alors compteur = 1)
- Mise à jour de la variable pour stocker le debut d'appui sur le bouton poussoir:
StartTime = millis()
bouton poussoir maintenu appuyé (broche à +5V):
ValButton = 1
et OldValButton = 1
?
Oui
- Mesure du temps d'appui sur le bouton poussoir:
DeltaTime = millis() - StartTime
- Si DeltaTime > 500 ms, le compteur d'appui sur le bouton poussoir est réinitialisé afin d'éteindre toutes les DELs:
ComptBtn = 0
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
En fonction de la valeur du compteur et du temps d'appui sur le bouton poussoir,
une DEL est allumée ou toutes les DELs sont éteintes:
if ComptBtn == 0 elif ComptBtn == 1 elif...
ComptBtn = 0
ComptBtn = 1
ComptBtn = 2
ComptBtn = 3
Eteindre toutes les DELs
board.digital[PinLEDR].write(0)
board.digital[PinLEDV].write(0)
board.digital[PinLEDB].write(0)
Allumer la DEL rouge et éteindre les autres DELs
board.digital[PinLEDR].write(1)
board.digital[PinLEDV].write(0)
board.digital[PinLEDB].write(0)
Allumer la DEL verte et éteindre les autres DELs
board.digital[PinLEDR].write(0)
board.digital[PinLEDV].write(1)
board.digital[PinLEDB].write(0)
Allumer la DEL bleue et éteindre les autres DELs
board.digital[PinLEDR].write(0)
board.digital[PinLEDV].write(0)
board.digital[PinLEDB].write(1)

2.5.2 DEL RVB : Entrées et Sorties analogiques

Après avoir vu le principe de fonctionnement des entrées et des sorties numériques de l'Arduino, avec ce nouveau circuit, nous allons nous intéresser aux entrées et sorties analogiques de la platine.



- Liste des composants :

. 1 DEL RVB
. 3 résistances de 220 Ω
. 2 résistances de 10 kΩ
. 3 potentiomètres de 10 kΩ
. 1 photorésistance
. 1 bouton poussoir
. 1 plaque d'essai
. Fils de connexion

- Protocole de communication (Mode "Contrôle de l'Arduino") :

. Firmata standard

Rappel :

. Les entrées analogiques

Il y a six entrées analogiques notées de A0 à A5 en bas à droite de la carte.

Les tensions, entre 0 et 5V, appliquées sur ces broches, sont numérisées via un convertisseur analogique-numérique CAN ou ADC (Analog Digital Converter).

Le convertisseur des Arduino effectue une conversion sur 10 bits, c'est à dire qu'il convertit la tension en un nombre entier ayant une valeur de 0 à 1023.

0 correspond à une tension de 0 V et 1023 à une tension de 5V.

La résolution, c'est à dire la différence entre deux valeurs successives de la tension correspondant à une différence de 1 sur l'entier résultat de la conversion analogique-numérique, est donc d'environ 5mV (**5/1024 = 4,9 mV**).

Attention :

. Appliquer une tension supérieure à 5 volts ou inférieure à 0 volt sur une broche analogique endommagera immédiatement et définitivement votre carte Arduino,

. La mesure prend environ 100µs, cela fait un maximum de 10 000 mesures par seconde,

. Effectuer une mesure d'une tension sur une broche non connectée retourne des valeurs de l'ordre de 300 à 500, même s'il n'y a pas de signal,

. La précision de la mesure (conversion sur 10 bits) n'est pas modifiable. Celle-ci est de plus ou moins 1 (+ ou - 5 mV).

Remarque :

Les broches notées de A0 à A5 peuvent également être configurées en entrées et sorties numériques.

. Les sorties analogiques (PWM)

Il n'y a pas de sortie analogique à proprement parler sur un Arduino. Par contre, six des connecteurs numériques (les connecteurs 3, 5, 6, 9, 10 et 11) sont capables de simuler des sorties analogiques et fonctionnent donc comme telles pour délivrer une tension entre 0 et 5 V.

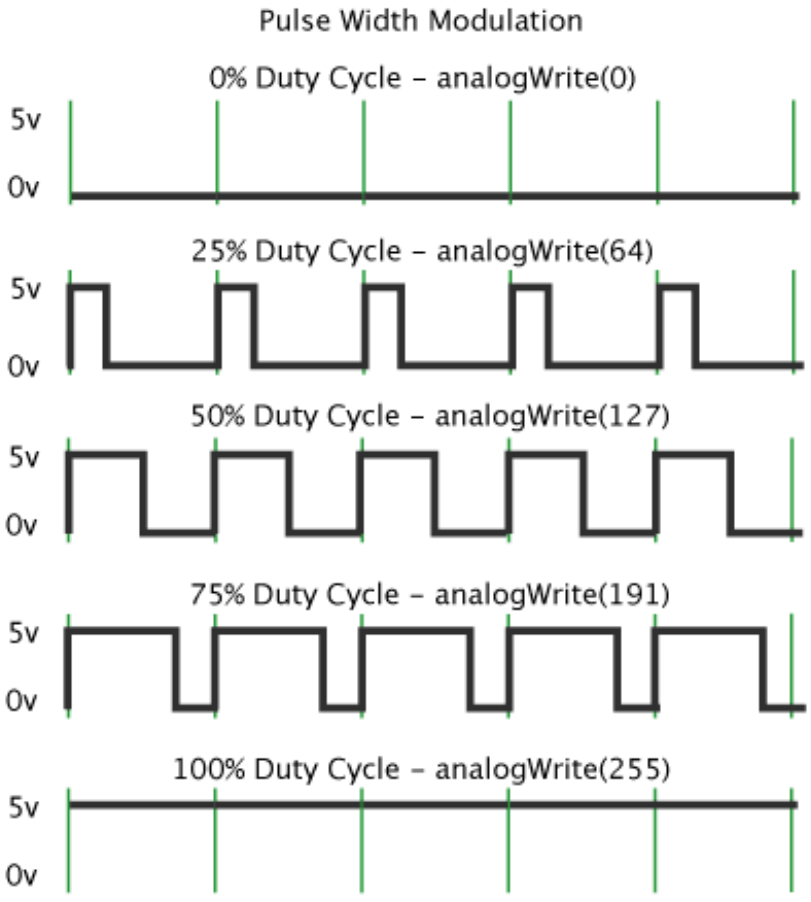
Elles sont marquées du symbole tilde ~ et du sigle PWM, qui veut dire Pulse Width Modulation (Modulation de Largeur d'Impulsion en français).

Le PWM fonctionne ainsi : comme il n'est possible que d'envoyer des informations binaires (haut ou bas, c'est à dire 5 V ou 0 V) sur ces broches, l'Arduino va faire varier la durée pendant laquelle ces deux valeurs sont appliquées afin d'obtenir la tension souhaitée.

L'Arduino génère donc un signal carré caractérisé par deux paramètres (Cf. ci-dessous) :

. L'amplitude du signal est de 5V ou de 0V,

. Le rapport entre la durée où la tension est à 5V et celle où elle est à 0V (ce rapport est appelé Duty cycle et est exprimé en %).



La fréquence du signal PWM est de 490 Hz, ce qui est suffisamment rapide pour que l'on puisse dire que l'amplitude du signal d'une sortie PWM est égale à la valeur moyenne du signal carré généré :

Tension sortie analogique (en V) = 5 x (Duty Cycle/100)

Exemple :

Duty Cycle à 0% : Tension sortie analogique = 5 x 0 = 0 V

Duty Cycle à 25% : Tension sortie analogique = 5 x 0,25 = 1,25 V

Duty Cycle à 50% : Tension sortie analogique = 5 x 0,5 = 2,5 V

Duty Cycle à 75% : Tension sortie analogique = 5 x 0,75 = 3,75 V

Duty Cycle à 100% : Tension sortie analogique = 5 x 1 = 5 V

Remarque :

En langage ARDUINO IDE, la fonction "**analogWrite ()**" permet de générer un signal analogique sur une sortie PWM.

Elle prend deux arguments :

. Le premier est le numéro de la broche sur laquelle on veut générer la PWM
. Le second argument représente la valeur du rapport cyclique à appliquer.

Cependant, on n'exprime pas cette valeur en pourcentage, mais avec un nombre entier compris entre 0 et 255.

Le rapport cyclique s'exprime de 0 à 100 % en temps normal. Cependant, dans cette fonction il s'exprimera de 0 à 255 (sur 8 bits). Ainsi, pour un rapport cyclique de 0% nous enverrons la valeur 0, pour un rapport de 50% on enverra 127 et pour 100% ce sera 255. Les autres valeurs sont bien entendu considérées de manière proportionnelle entre les deux.

- Activité 1 : Contrôler la luminosité d'une DEL avec un bouton-poussoir

Contrairement aux sorties numériques qui ne peuvent avoir que deux valeurs 0 ou 1 (0 ou 5V), une sortie analogique (ou plutôt PWM) permet d'obtenir une tension entre 0 et 5 V. les broches 3, 5, 6, 9, 10 et 11 peuvent être configurés en sortie analogique.

C'est pour cela que les anodes de la DEL RVB (à cathode commune) de notre circuit sont connectées sur les broches :

- 9 pour la DEL rouge
- 11 pour la DEL verte
- 10 pour la DEL Bleue

Nous allons utiliser ces sorties pour alimenter une DEL (DEL rouge, verte ou bleue de la DEL RVB) et faire varier sa luminosité suivant ce principe de fonctionnement :

- la DEL étant éteinte, si on appuie sur le bouton poussoir, la diode s'allume.

- On règle la luminosité de la DEL en maintenant le bouton poussoir appuyé, du moins au plus lumineux (broche de la DEL à +0V) jusqu'à un maximum (broche de la DEL à +5V).

- Quand le maximum de la luminosité est atteint et que le bouton poussoir est maintenu appuyé, la luminosité revient au minimum (broche de la DEL à 0V).

- La DEL étant allumée, si on appuie sur le bouton poussoir, elle s'éteint.

Après avoir cliqué sur le connecteur USB, le choix de la DEL est fait par l'intermédiaire de ce menu :

Une modification dans le choix de la DEL quand une DEL est déjà allumée, entraine une réinitialisation du programme. Il faut appuyer de nouveau sur le bouton poussoir pour allumer la DEL choisie.

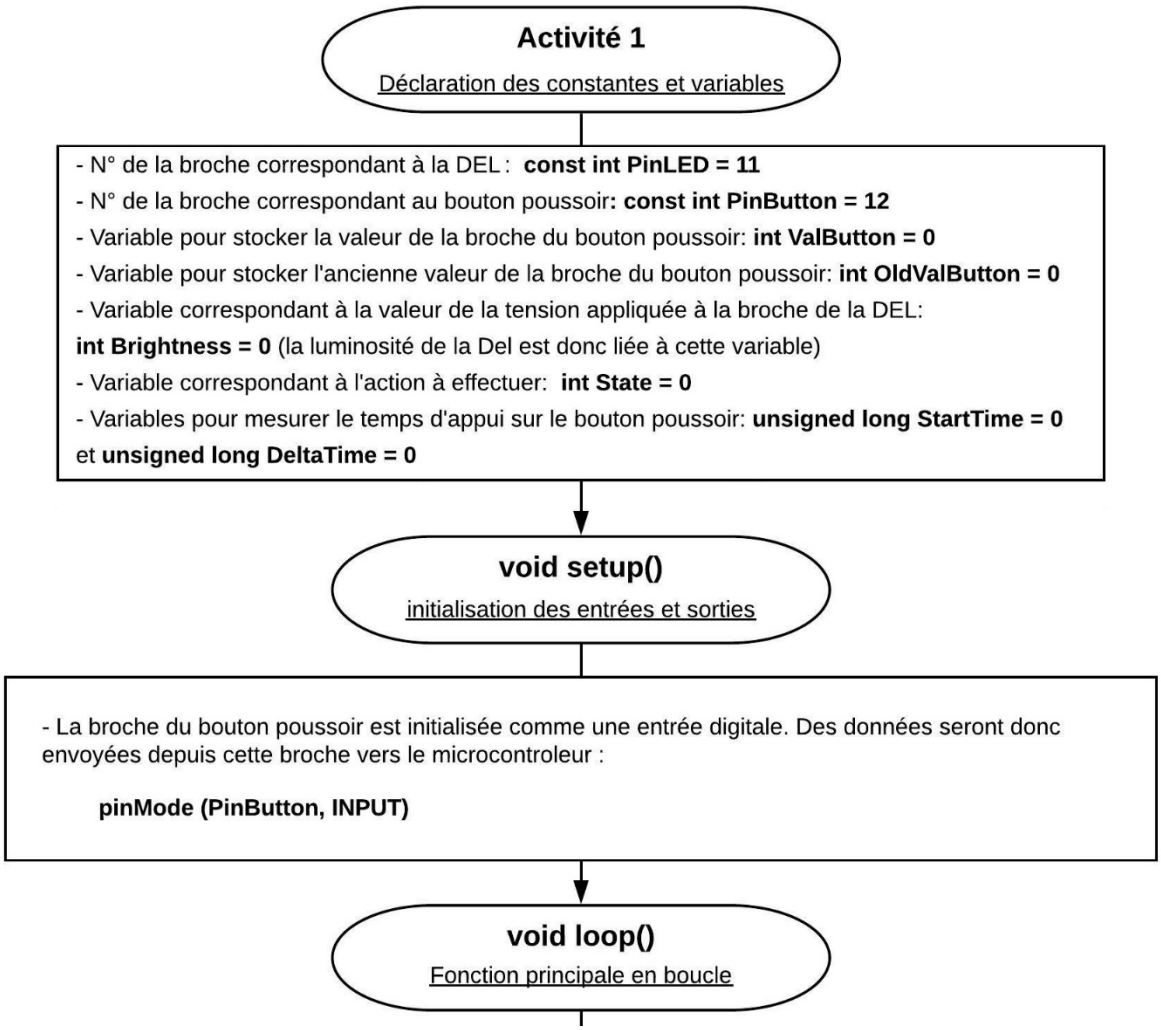
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DELs sur l'écran s'allument ou s'éteignent en appuyant sur le bouton poussoir du circuit réel ou sur celui du circuit affiché sur l'écran.

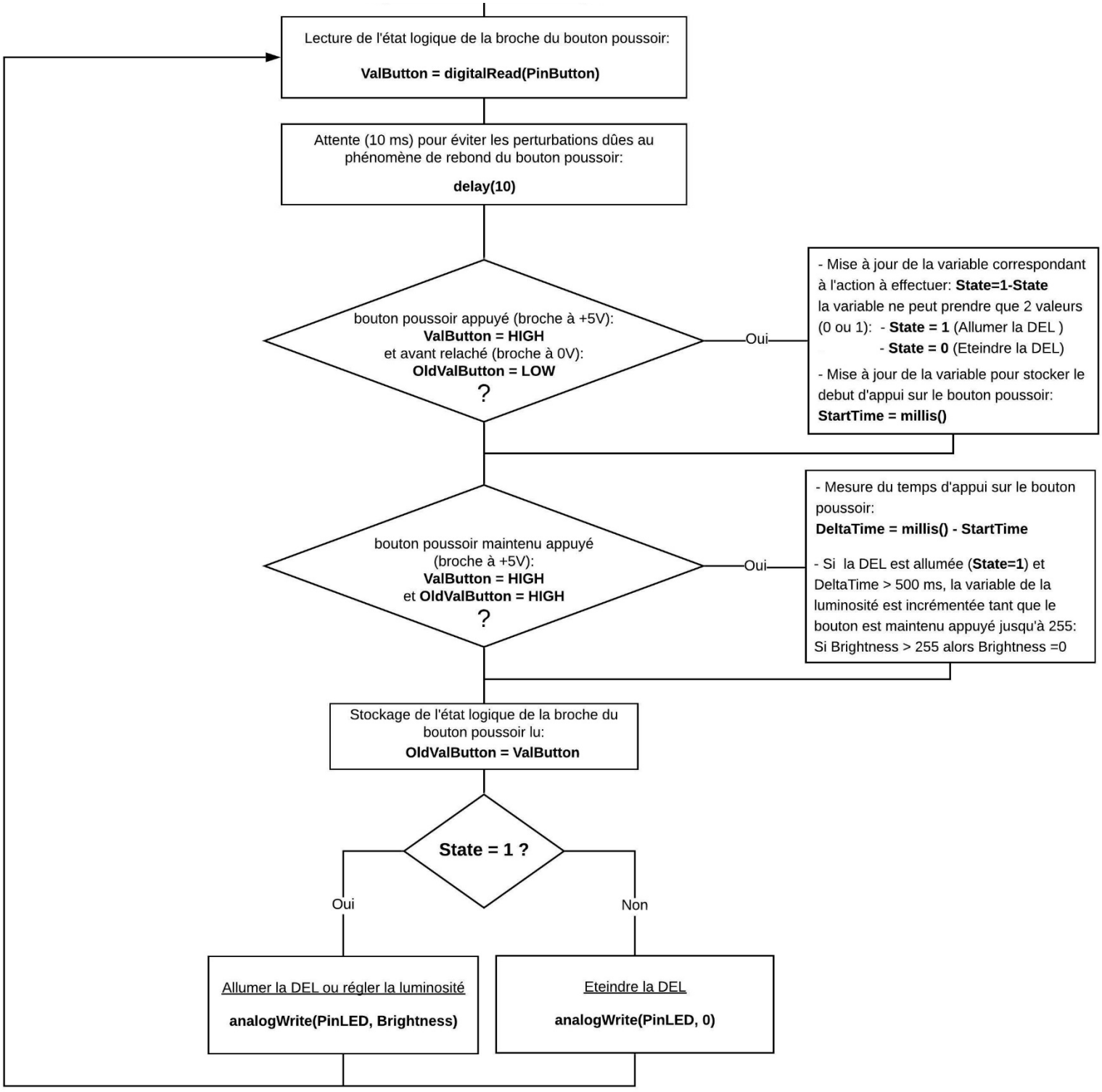
A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

Le code pourra être modifié pour voir l'influence des variables (choix de la DEL).

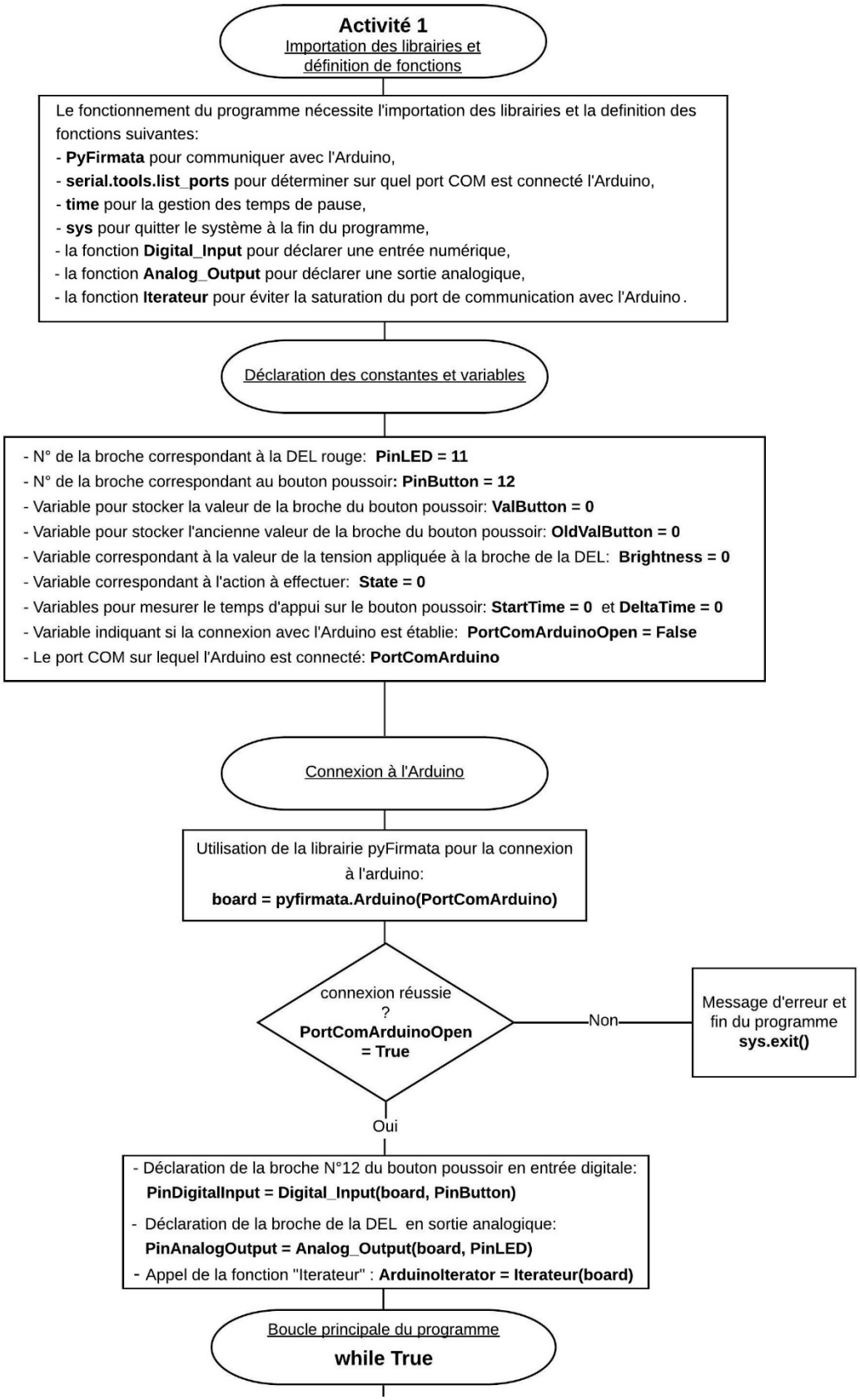
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

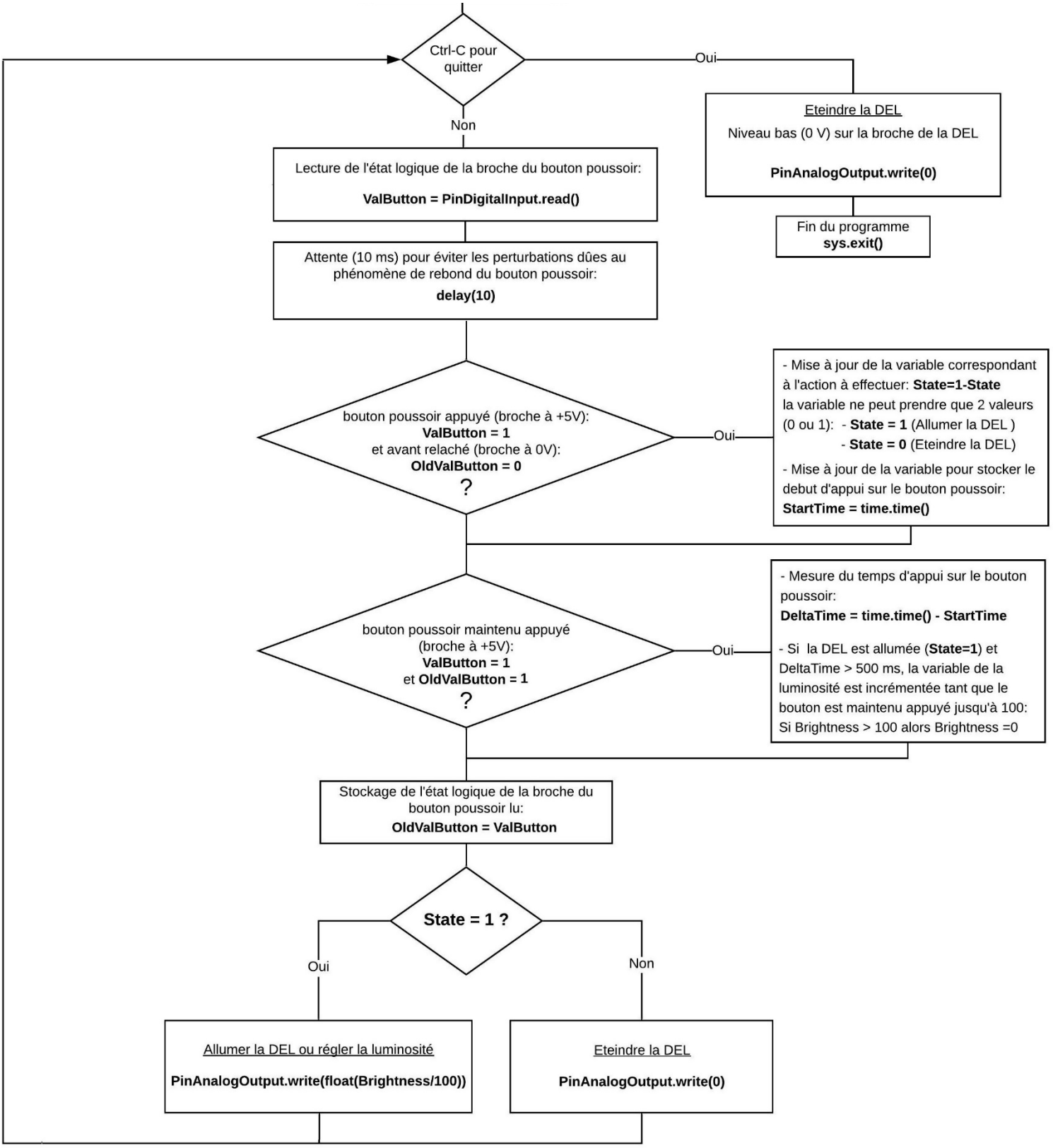
* Algorithme de programmation de l'activité 1 en langage Arduino IDE :




Lecture de l'état logique de la broche du bouton poussoir:
ValButton = digitalRead(PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = HIGH
et avant relaché (broche à 0V):
OldValButton = LOW
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1): - State = 1 (Allumer la DEL)
- State = 0 (Eteindre la DEL)
- Mise à jour de la variable pour stocker le debut d'appui sur le bouton poussoir:
StartTime = millis()
bouton poussoir maintenu appuyé (broche à +5V):
ValButton = HIGH
et OldValButton = HIGH
?
Oui
- Mesure du temps d'appui sur le bouton poussoir:
DeltaTime = millis() - StartTime
- Si la DEL est allumée (State=1) et DeltaTime > 500 ms, la variable de la luminosité est incrémentée tant que le bouton est maintenu appuyé jusqu'à 255:
Si Brightness > 255 alors Brightness =0
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Allumer la DEL ou régler la luminosité
analogWrite(PinLED, Brightness)
Eteindre la DEL
analogWrite(PinLED, 0)

* Algorithme de programmation de l'activité 1 en Python :




Ctrl-C pour quitter
Oui
Non
Eteindre la DEL
Niveau bas (0 V) sur la broche de la DEL
PinAnalogOutput.write(0)
Fin du programme
sys.exit()
Lecture de l'état logique de la broche du bouton poussoir:
ValButton = PinDigitalInput.read()
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1): - State = 1 (Allumer la DEL)
- State = 0 (Eteindre la DEL)
- Mise à jour de la variable pour stocker le debut d'appui sur le bouton poussoir:
StartTime = time.time()
bouton poussoir maintenu appuyé (broche à +5V):
ValButton = 1
et OldValButton = 1
?
Oui
- Mesure du temps d'appui sur le bouton poussoir:
DeltaTime = time.time() - StartTime
- Si la DEL est allumée (State=1) et DeltaTime > 500 ms, la variable de la luminosité est incrémentée tant que le bouton est maintenu appuyé jusqu'à 100:
Si Brightness > 100 alors Brightness =0
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Allumer la DEL ou régler la luminosité
PinAnalogOutput.write(float(Brightness/100))
Eteindre la DEL
PinAnalogOutput.write(0)

- Activité 2 : Clignotement d'une DEL à une allure fixée par une entrée analogique

Maintenant que nous connaissons le fonctionnement des sorties analogiques, nous allons aborder, dans cette activité, l'étude des entrées analogiques de l'Arduino.

Ces entrées (A0 à A5), qui peuvent être également utilisées en entrées digitales, sont capables de mesurer la tension réelle, entre 0 et 5V, qui leur est appliquée. On utilisera ces entrées pour les acquisitions avec des capteurs qui délivrent une tension entre 0 et 5 V suivant ce qu'ils mesurent.

Ici, le capteur utilisé est une photorésistance, dont la résistance varie en fonction de l'intensité lumineuse qu'elle reçoit. La sortie de la photorésistance est branchée sur une des entrées analogiques de la carte Arduino (entrée A5).

L'objectif est de faire clignoter une DEL à une fréquence dépendant de la lumière ambiante. Pour cela, on va faire varier le délai entre 2 allumages de la DEL en fonction de la tension de l'entrée A5 et donc de l'intensité lumineuse reçue par la photorésistance.
Quand l'intensité lumineuse reçue par la photorésistance diminue, la fréquence de clignotement augmente

Après avoir cliqué sur le connecteur USB, le choix de la DEL est fait par l'intermédiaire de ce menu :

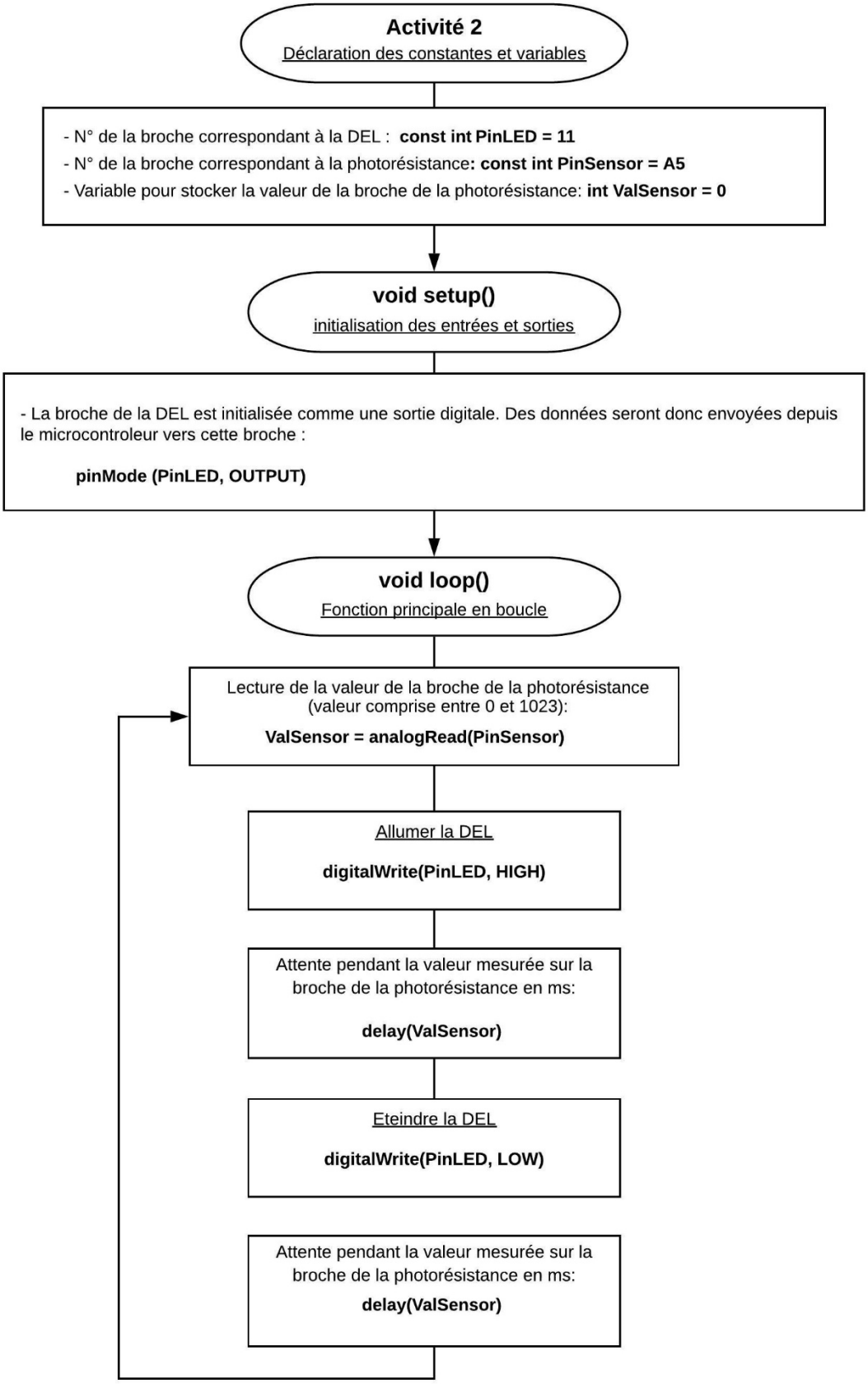
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DELs sur l'écran clignotent à une fréquence dépendant de la valeur de la photorésistance réelle ou de celle du circuit sur l'écran (le changement d'intensité lumineuse est simulé en passant la souris sur la photorésistance).

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

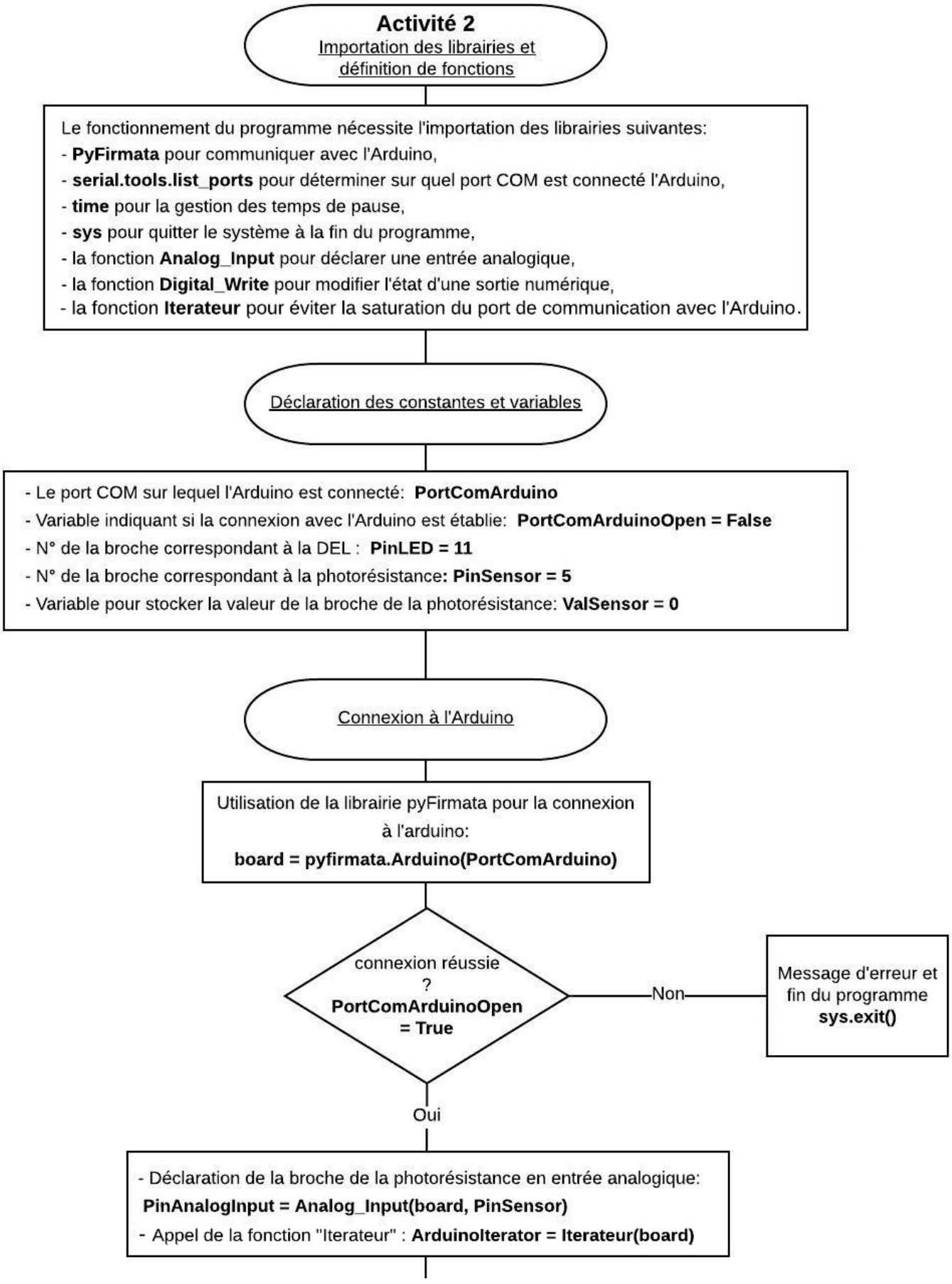
Le code pourra être modifié pour voir l'influence des variables (choix de la DEL).

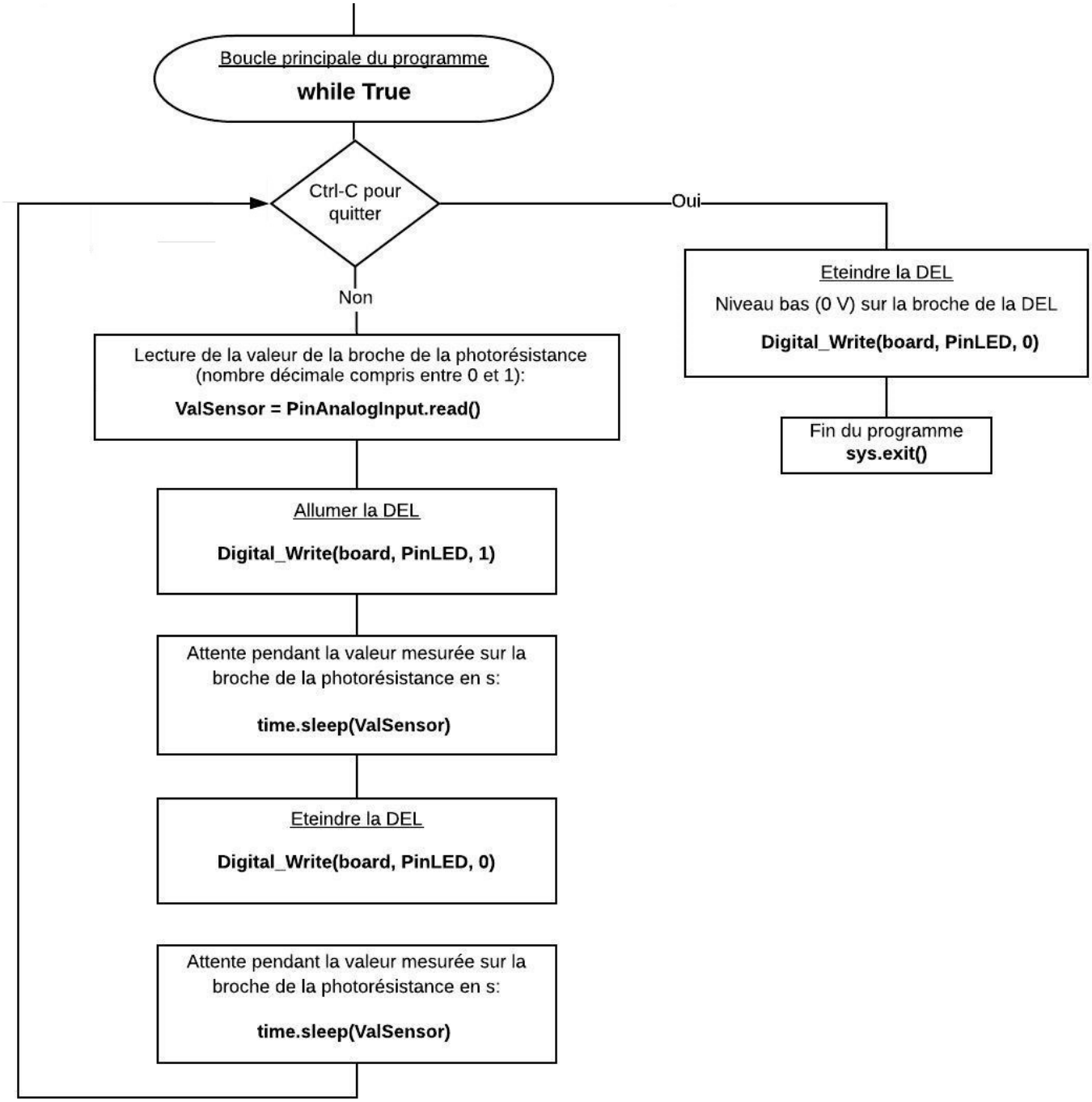
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 2 en langage Arduino IDE :



* Algorithme de programmation de l'activité 2 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre la DEL
Niveau bas (0 V) sur la broche de la DEL
Digital_Write(board, PinLED, 0)
Fin du programme
sys.exit()
Lecture de la valeur de la broche de la photorésistance (nombre décimale compris entre 0 et 1):
ValSensor = PinAnalogInput.read()
Allumer la DEL
Digital_Write(board, PinLED, 1)
Attente pendant la valeur mesurée sur la broche de la photorésistance en s:
time.sleep(ValSensor)
Eteindre la DEL
Digital_Write(board, PinLED, 0)
Attente pendant la valeur mesurée sur la broche de la photorésistance en s:
time.sleep(ValSensor)

- Activité 3 : Réglage de la luminosité d'une DEL à un niveau fixé par une entrée analogique

Dans cette activité, selon le même principe que l'activité précédente, on va faire varier la luminosité d'une DEL en fonction de l'intensité lumineuse reçue par la photorésistance (la luminosité de la DEL est inversement proportionnelle à l'intensité lumineuse reçue) :

- La DEL est allumée ou éteinte en appuyant sur le bouton-poussoir,
- La luminosité de la DEL varie en fonction de la tension de l'entrée A5.

Après avoir cliqué sur le connecteur USB, le choix de la DEL est fait par l'intermédiaire de ce menu :

Une modification dans le choix de la DEL quand une DEL est déjà allumée, entraine une réinitialisation du programme. Il faut appuyer de nouveau sur le bouton poussoir pour allumer la DEL choisie.

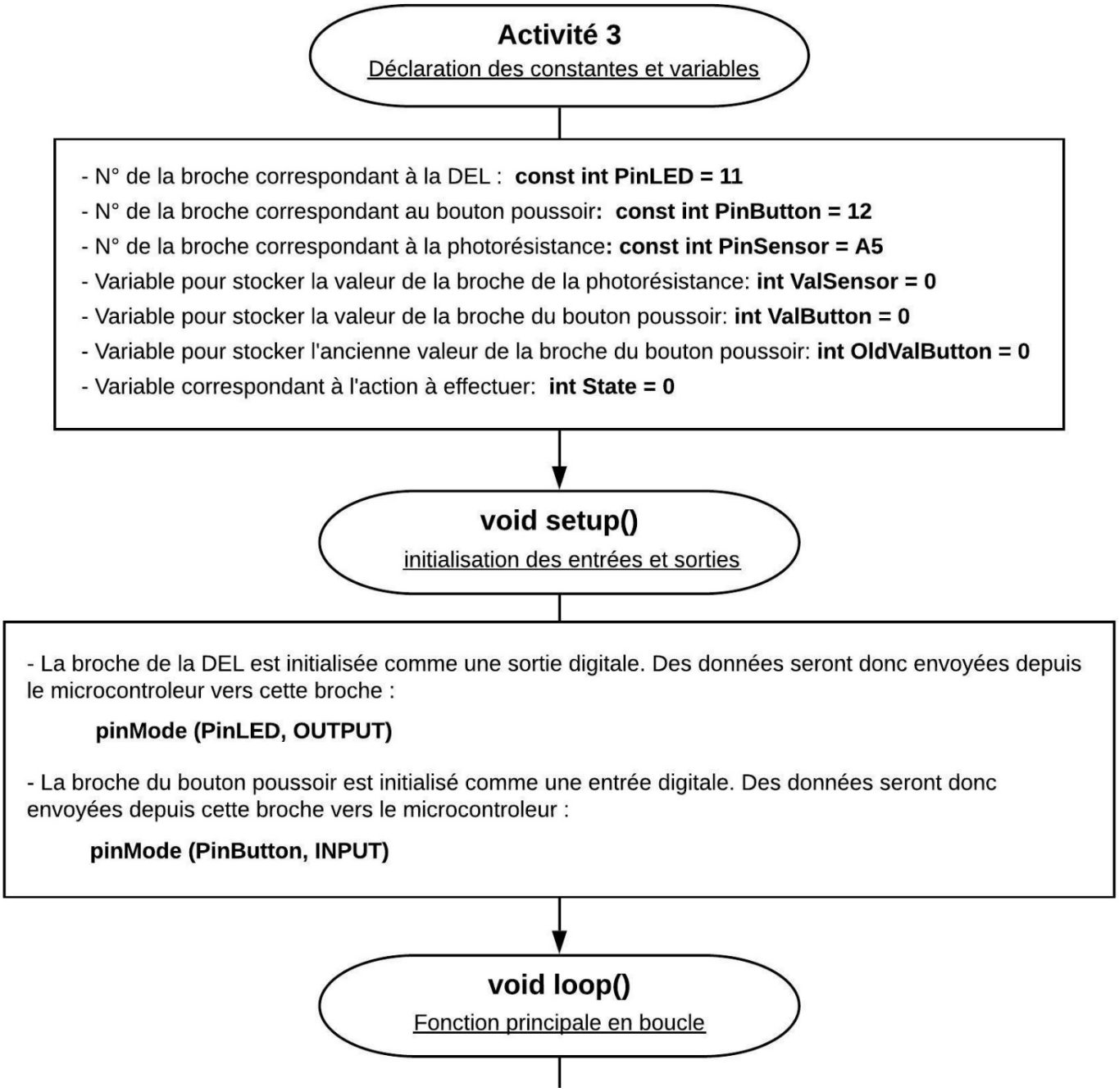
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DEL sur l'écran s'allument ou s'éteignent en appuyant sur le bouton poussoir du circuit réel ou sur celui du circuit affiché sur l'écran. La luminosité de la DEL varie en fonction de la valeur de la photorésistance réelle ou de celle du circuit sur l'écran (le changement d'intensité lumineuse est simulé en passant la souris sur la photorésistance).

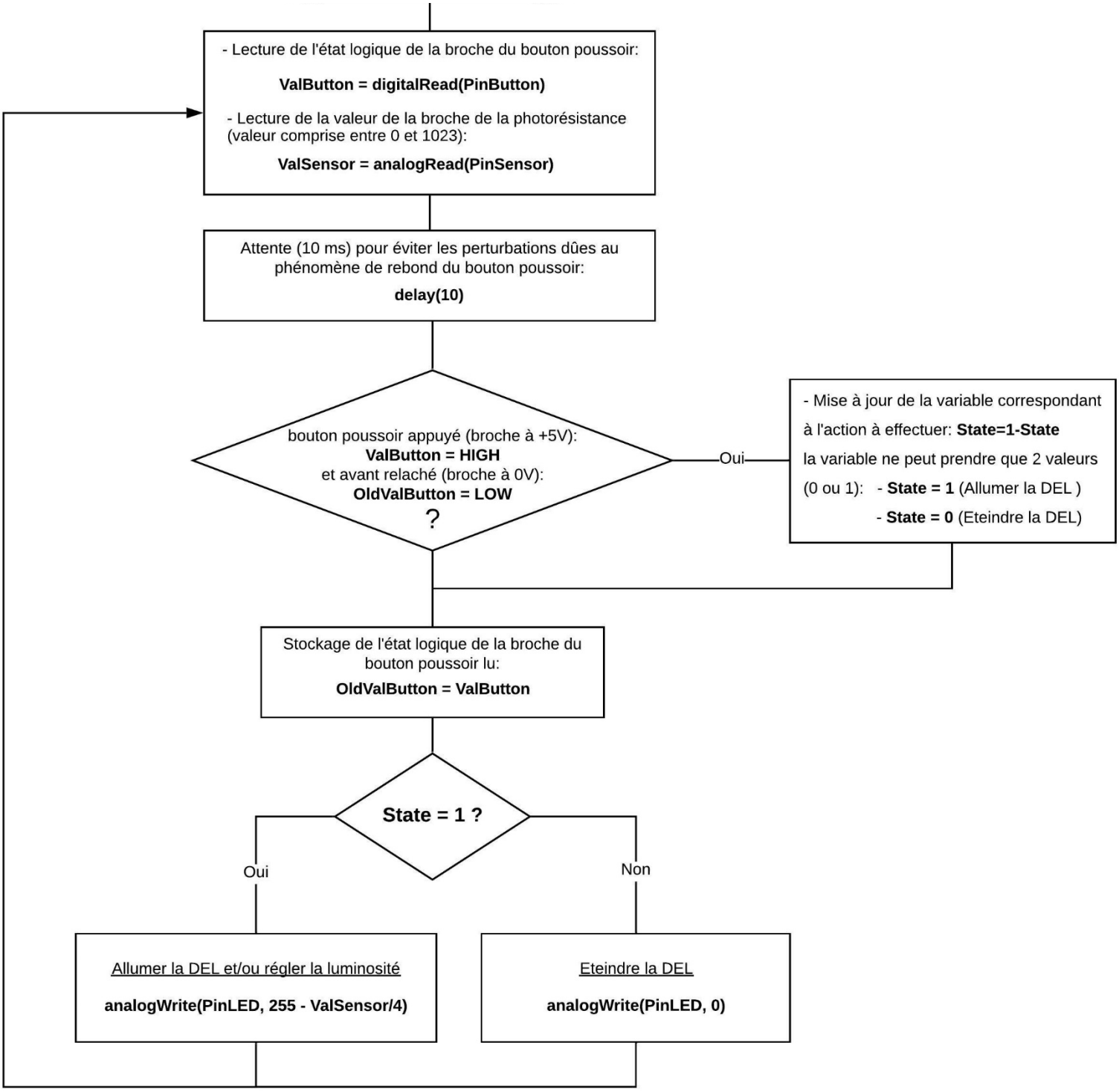
A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

Le code pourra être modifié pour voir l'influence des variables (choix de la DEL).

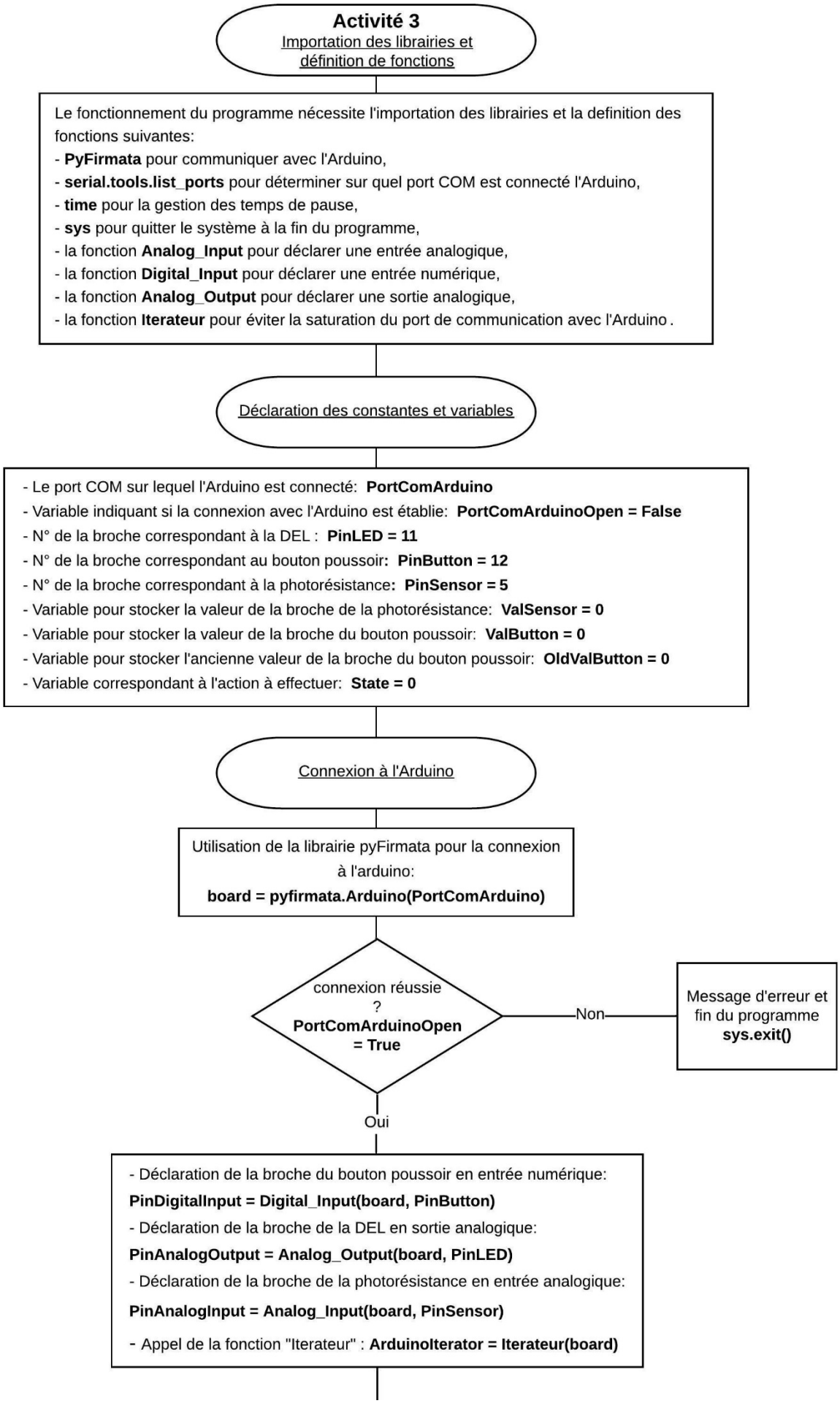
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

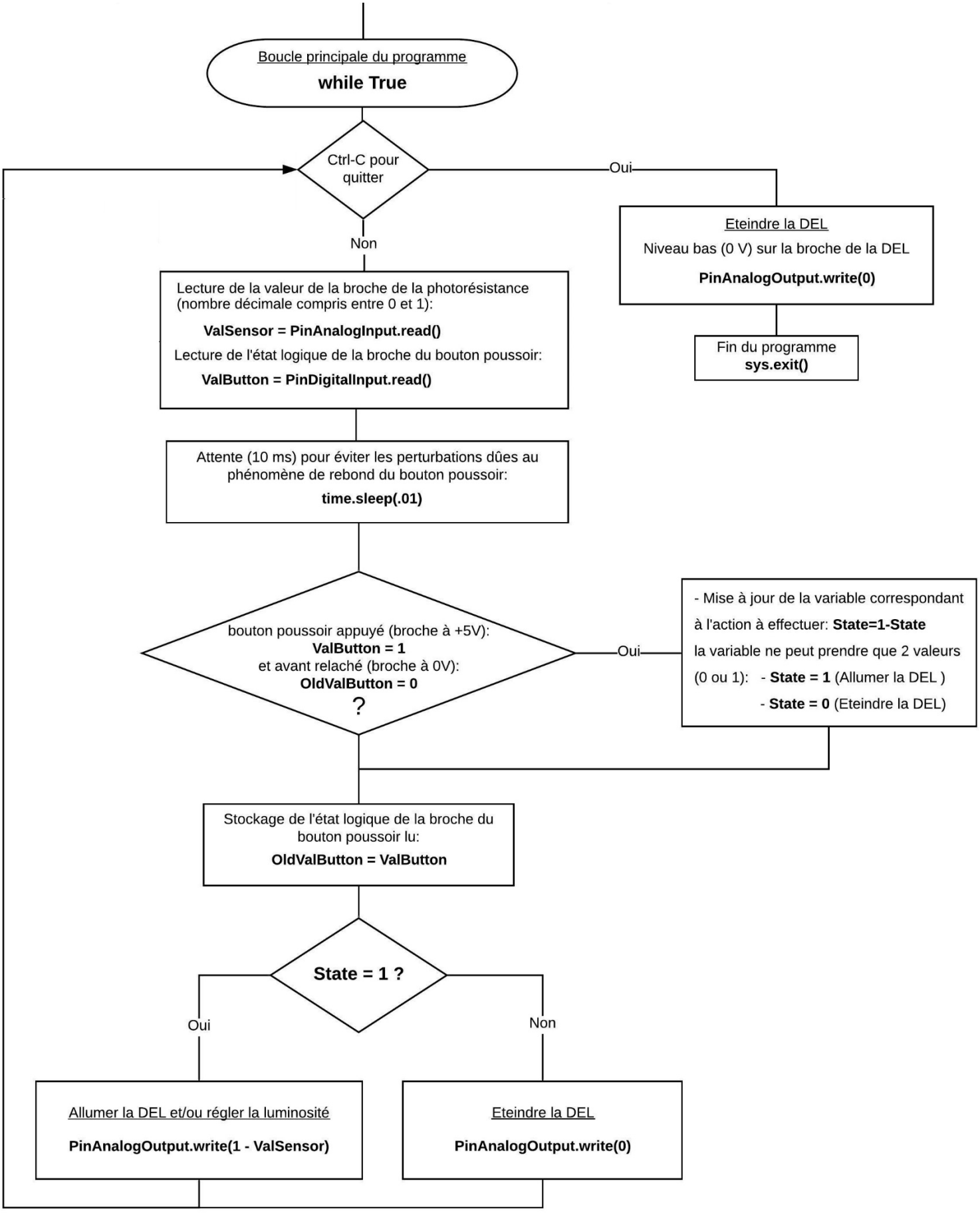
* Algorithme de programmation de l'activité 3 en langage Arduino IDE :




- Lecture de l'état logique de la broche du bouton poussoir:
ValButton = digitalRead(PinButton)
- Lecture de la valeur de la broche de la photorésistance (valeur comprise entre 0 et 1023):
ValSensor = analogRead(PinSensor)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = HIGH
et avant relaché (broche à 0V):
OldValButton = LOW
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1): - State = 1 (Allumer la DEL)
- State = 0 (Eteindre la DEL)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Allumer la DEL et/ou régler la luminosité
analogWrite(PinLED, 255 - ValSensor/4)
Eteindre la DEL
analogWrite(PinLED, 0)

* Algorithme de programmation de l'activité 3 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre la DEL
Niveau bas (0 V) sur la broche de la DEL
PinAnalogOutput.write(0)
Fin du programme
sys.exit()
Lecture de la valeur de la broche de la photorésistance (nombre décimale compris entre 0 et 1):
ValSensor = PinAnalogInput.read()
Lecture de l'état logique de la broche du bouton poussoir:
ValButton = PinDigitalInput.read()
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(.01)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1): - State = 1 (Allumer la DEL)
- State = 0 (Eteindre la DEL)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Allumer la DEL et/ou régler la luminosité
PinAnalogOutput.write(1 - ValSensor)
Eteindre la DEL
PinAnalogOutput.write(0)

- Activité 4 : DEL RVB - Synthèse additive des couleurs

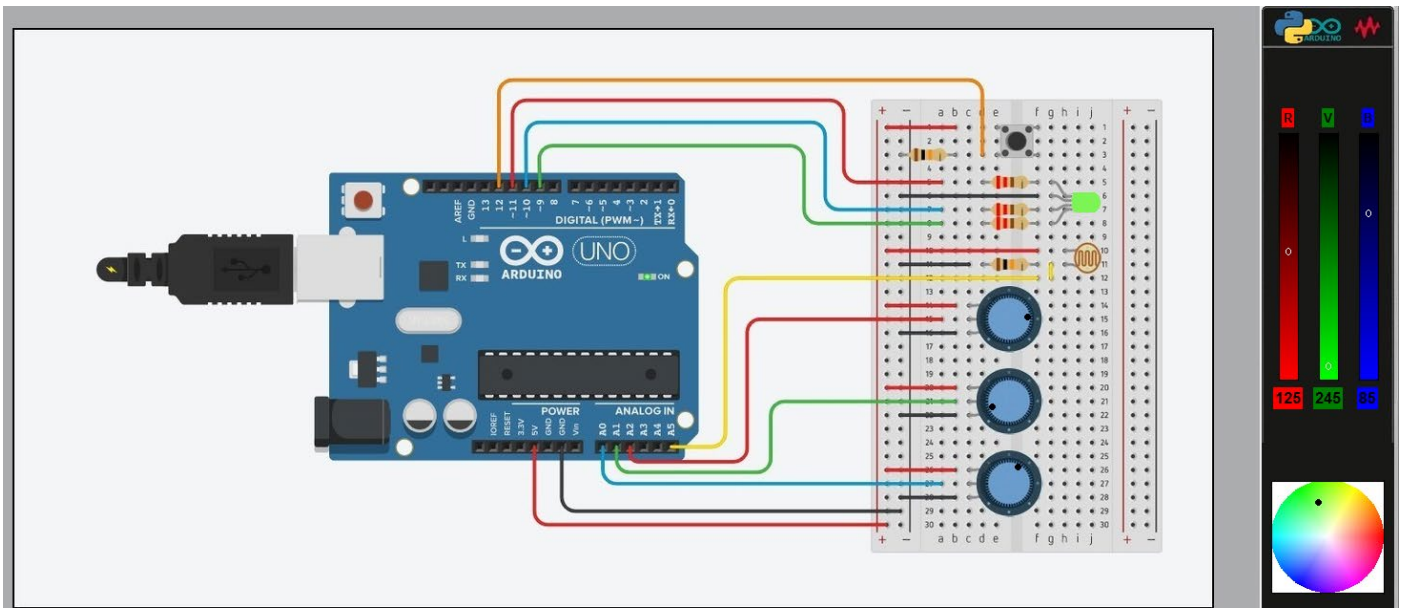
Cette activité est une application directe des applications précédentes. Nous allons utiliser 3 potentiomètres respectivement connectés aux entrées analogiques A2, A1 et A0 de l'Arduino pour régler les luminosités des DELs Rouge, Verte et Bleue d'une DEL RVB.

En effet, celle-ci dispose de 4 broches, 1 cathode et 3 anodes. Chaque anode correspond à une couleur (Rouge, Vert et Bleu).
En modulant les signaux sur les anodes, il est possible d'obtenir de multiples couleurs. C'est le principe de la synthèse additive des couleurs des pixels des écrans d'ordinateur ou de télévision.

- Dans un premier temps, la DEL RVB est allumée en appuyant sur le bouton poussoir,
- La luminosité des DELs varie en fonction de la tension des entrées A2, A1 et A0,
- La DEL RVB est éteinte en appuyant de nouveau sur le bouton poussoir.

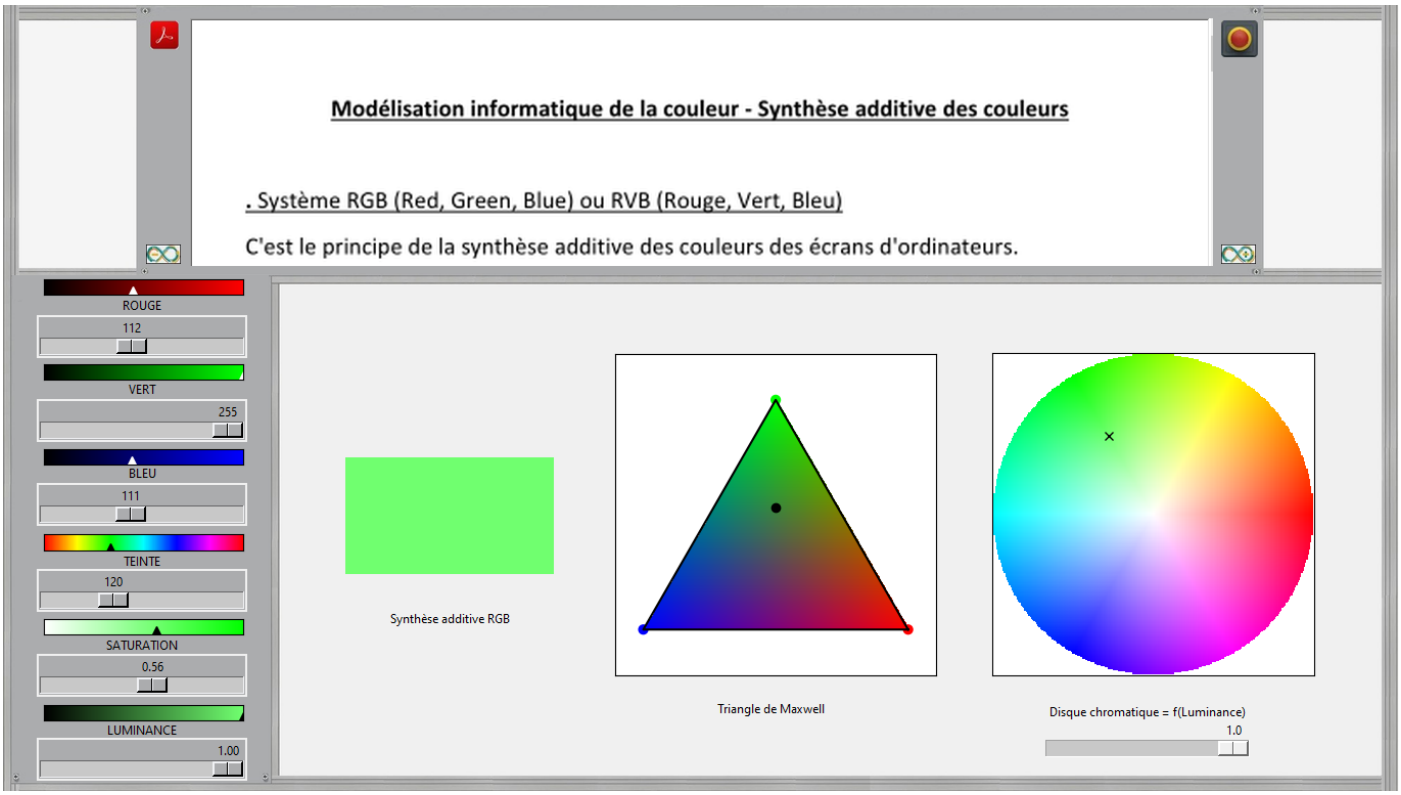
Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL RVB du circuit réel et la DEL RVB sur l'écran s'allument ou s'éteignent en appuyant sur le bouton poussoir du circuit réel ou sur celui du circuit affiché sur l'écran. La luminosité des DELs (réelle et virtuelle) est réglée, soit à l'aide des potentiomètres réels, soit à l'aide des potentiomètres du circuit sur l'écran (**la molette de la souris permet de régler le potentiomètre quand la souris est dessus**).

Après avoir cliqué sur le connecteur USB, un menu avec les valeurs des composantes Rouge, Verte et Bleue de la couleur produite par synthèse additive et son emplacement sur un disque chromatique est affiché :



En cliquant sur le disque chromatique, une nouvelle fenêtre expliquant le principe de la synthèse additive RVB (Rouge, Vert, Bleu) s'ouvre.

Dans cette fenêtre, il est possible de modifier les valeurs des composantes RVB, de voir le déplacement de la couleur produite sur le disque chromatique, de choisir une couleur sur celui-ci et de connaitre ses composantes RVB :



Rappel sur la modélisation informatique de la couleur - Synthèse additive des couleurs

. Système RGB (Red, Green, Blue) ou RVB (Rouge, Vert, Bleu)

C'est le principe de la synthèse additive des couleurs des écrans d'ordinateurs.

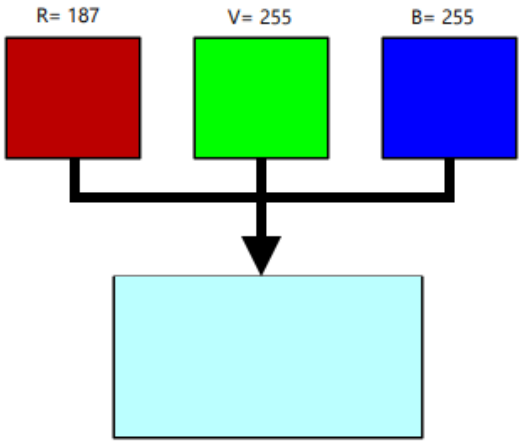
Toute couleur est obtenue en ajoutant différentes quantités de rouge, de vert et de bleu qui sont les seules couleurs dont on dispose à la base.

En effet, Les écrans d'ordinateur (cathodique, LCD), et d'une manière générale les systèmes de formation d'image numérique, fonctionnent sur le principe du mélange additif. Chaque pixel de l'écran est constitué de trois cellules, une verte, une bleue et une rouge. L'intensité lumineuse émise par ces cellules est ajustée pour produire la couleur voulue

Les images numériques destinées à l'affichage sur ces écrans sont codées en RVB. Elles comportent une couche rouge, une couche verte et une couche bleue. Chaque couche, le plus souvent codée sur 8 bits (valeurs de 0 à 255) représente le niveau d'intensité qui doit être délivré par la cellule correspondante sur l'écran.

Ces quantités de rouge, de vert et de bleu peuvent être exprimées sous forme de pourcentage (entre 0 et 1) ou sous forme de nombres (généralement compris entre 0 et 255) :

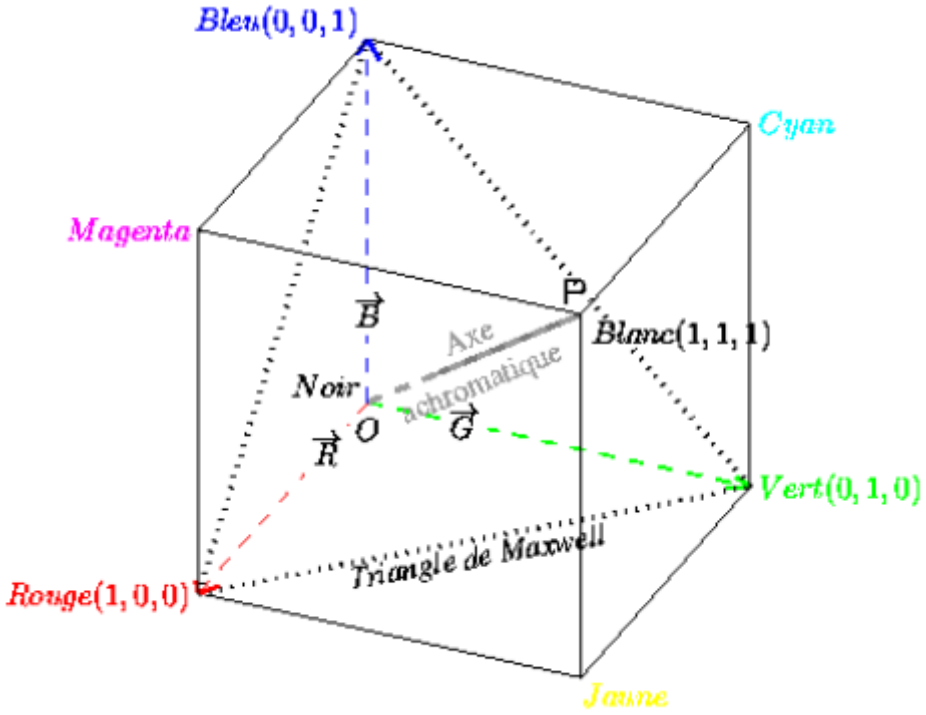
	R	V	B
Noir	0	0	0
Bleu	0	0	255
Vert	0	255	0
Cyan	0	255	255
Rouge	255	0	0
Magenta	255	0	255
Jaune	255	255	0
Blanc	255	255	255



Si on note R, V et B les trois couleurs primaires utilisées, une couleur C peut s'écrire comme la combinaison linéaire :

$\vec{C} = r\vec{R} + v\vec{V} + b\vec{B}$ (r, v, b sont les intensités relatives variables des trois couleurs)

Le système RGB peut être représenté sous la forme d'un cube (espace des couleurs et triangle de Maxwell) :

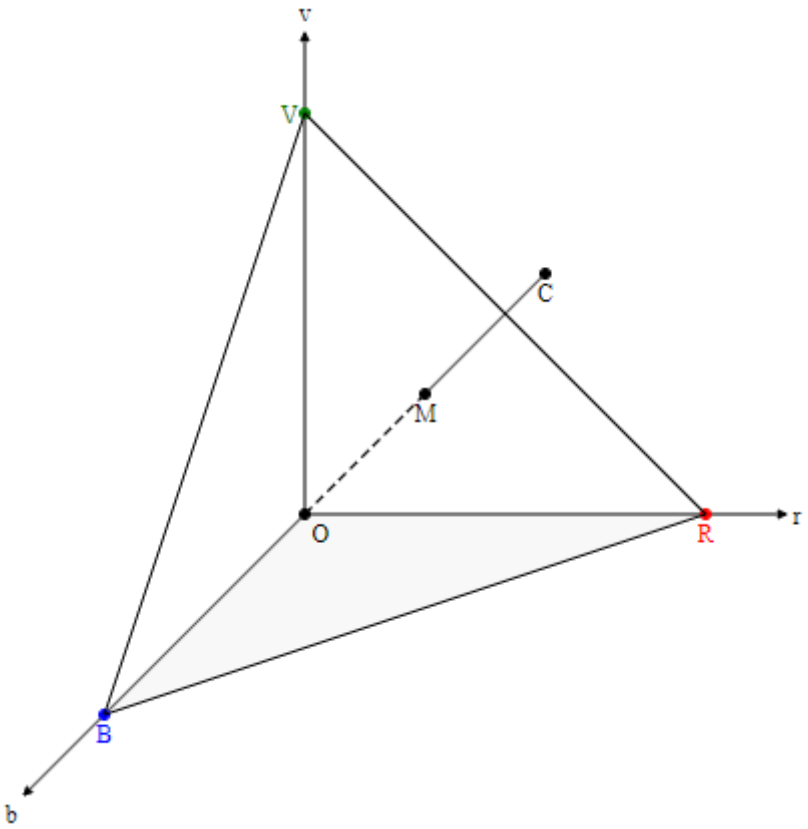


Les coefficients réels (r, v, b) sont positifs. Par convention, on peut les choisir dans l'intervalle [0,1].

Pour représenter géométriquement les couleurs, on représente les primaires par une base orthonormée et on place le noir à l'origine O. L'ensemble des couleurs est alors contenu dans un cube de côté 1. Le point P (1,1,1) est le sommet du cube opposé à O.

Soit C (r, g, b) un point représentant une couleur. Lorsqu'on multiplie tous les coefficients (r, g, b) par une même constante, on ne change pas la qualité d'une couleur, que l'on appelle sa chromaticité, mais seulement sa luminosité. Ainsi, toutes les couleurs de la droite OC sont perçues avec la même chromaticité. En particulier, les points de la diagonale OP sont des points neutres, c'est-à-dire des gris, obtenus par un mélange égal des trois primaires. Le point O (0,0,0) est le noir, le point P (1,1,1) est le blanc.

Considérons le plan d'équation r+v+b=1 qui passe par les points R (1,0,0), V (0,1,0) et B (0,0,1). Ces 3 points définissent dans ce plan un triangle appelé triangle de Maxwell. L'intersection de la droite OC avec ce triangle est le point M. La position de ce point dans le triangle suffit à déterminer la chromaticité de la couleur. D'un point de vue chromatique, toutes les couleurs sont donc représentables dans le triangle de Maxwell. Les coordonnées trichromatiques (r, v, b) vérifient alors r+v+b=1.

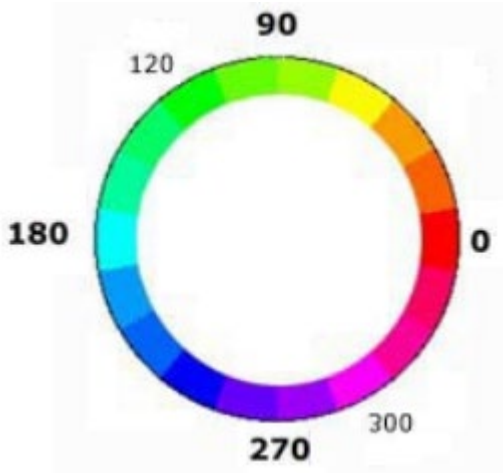


. Système TSL (ou HSL)

Toute couleur est décrite par :

- une Teinte (Hue)
- une Saturation (Saturation)
- une Luminance (Luminance)

La teinte permet de déterminer la couleur souhaitée à partir des couleurs rouge, vert et bleu. Elle est exprimée par un nombre qui est sa position angulaire sur le cercle chromatique. La teinte est donc un angle dans l'intervalle [0,360] :



Ex : rouge : 0 ; vert : 120 ; magenta : 300.

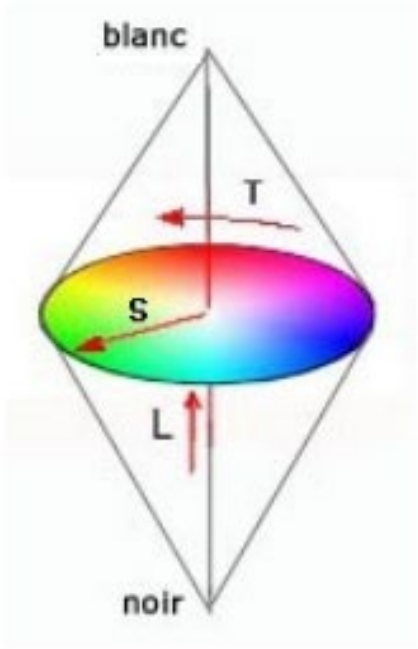
La saturation mesure l'intensité ou la pureté d'une couleur, c'est-à-dire le pourcentage de couleur pure par rapport au blanc. La saturation permet donc de distinguer une couleur vive d'une couleur pastel.

La saturation est représentée sur le rayon du cercle, par un pourcentage de pureté : elle est maximale sur le cercle (100% ou 1) et minimale au centre (0 = gris)

La luminance permet de définir la part de noir ou de blanc dans la couleur désirée (couleur claire ou sombre).

L'ensemble des couleurs est représenté à l'intérieur d'un double cône. La luminance varie sur l'axe vertical du double cône (axe des gris) du noir en bas au blanc, en haut.

La luminance est exprimée par un pourcentage : de 0% (noir) à 100% (blanc).



En résumé :

- Une augmentation de la luminance d'une couleur la fait tendre vers le blanc.
- Une diminution de sa luminance la fait tendre vers le noir.
- Une diminution de la saturation fera tendre cette couleur vers le gris (axe du double cône).

. Conversion RGB/TSL

On définit la luminance comme le maximum de r, g, b. Voici, l'algorithme qui permet de calculer la teinte (T), la saturation (S) et la luminance (L) :

$$Max = max(r, g, b)$$
$$Min = min(r, g, b)$$
$$C = Max - Min$$
$$L = Max$$
$$S = \frac{C}{L}$$
$$T = 60\frac{v - b}{C} \ [360] \ \ si \ \ Max = r$$
$$T = 120 + 60\frac{b - r}{C} \ \ si \ \ Max = v$$
$$T = 240 + 60\frac{r - v}{C} \ \ si \ \ Max = b$$

Si C=0, il s'agit d'une couleur neutre (trois couleurs égales) pour laquelle la teinte n'est pas définie et la saturation nulle. Les valeurs de S et L sont dans l'intervalle [0,1]. La teinte est un angle dans l'intervalle [0,360].

. Conversion TSL/RGB

D’après l'algorithme de conversion RGB/TSL, on a C=LS et Min=L-C et suivant l’intervalle dans lequel la teinte T se trouve, on calcul les valeurs de r, v, et b :

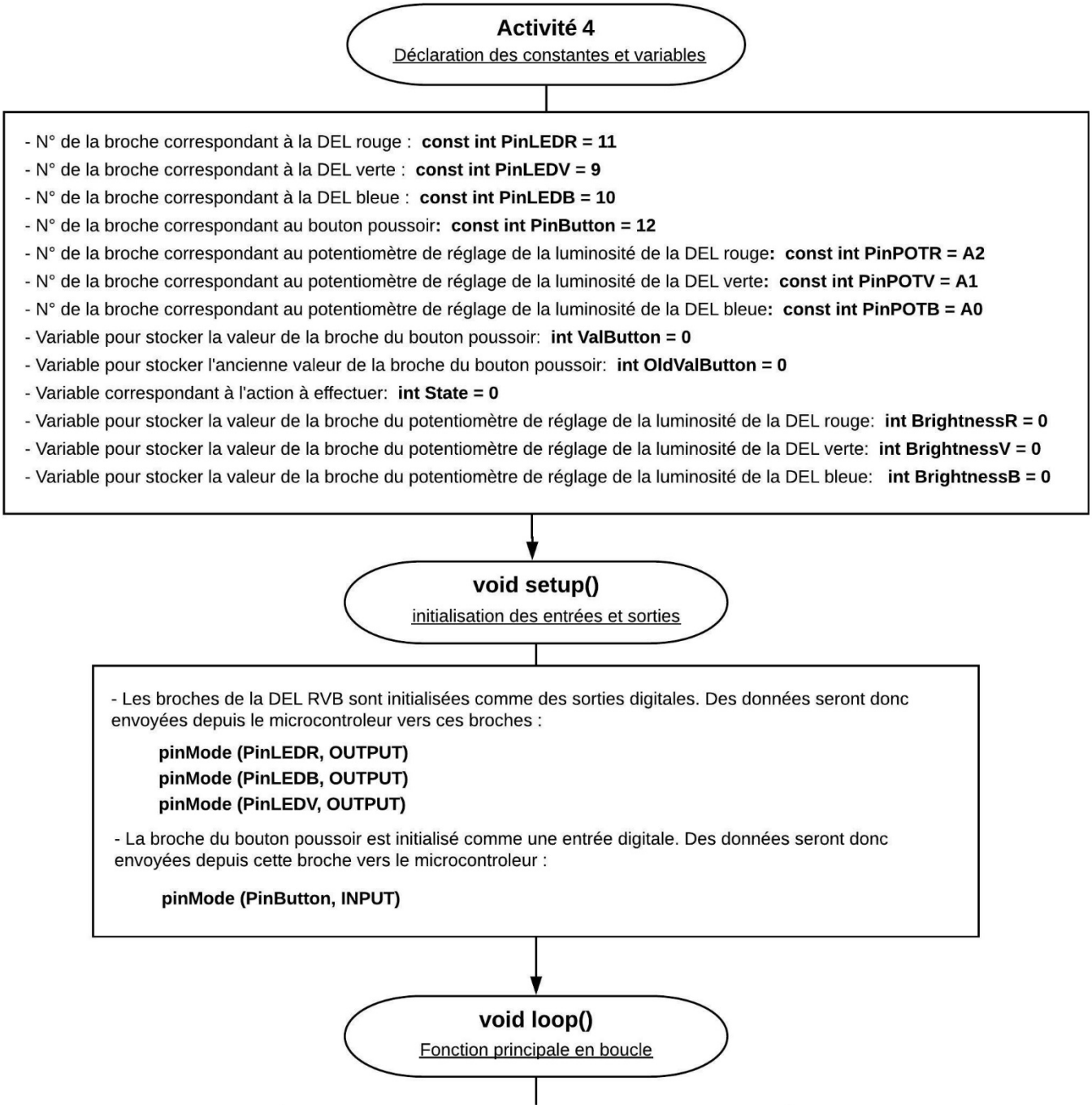
$T \in 300, 360$	$r = L$	$v = L - C$	$b = v + C(360 - T)/60$
$T \in 0, 60$	$r = L$	$b = L - C$	$v = b + C(T/60)$
$T \in 60, 120$	$v = L$	$b = L - C$	$r = b + C(120 - T)/60$
$T \in 120, 180$	$v = L$	$r = L - C$	$b = r + C(T - 120)/60$
$T \in 180, 240$	$b = L$	$r = L - C$	$v = r + C(240 - T)/60$
$T \in 240, 300$	$b = L$	$v = L - C$	$r = v + C(T - 240)/60$

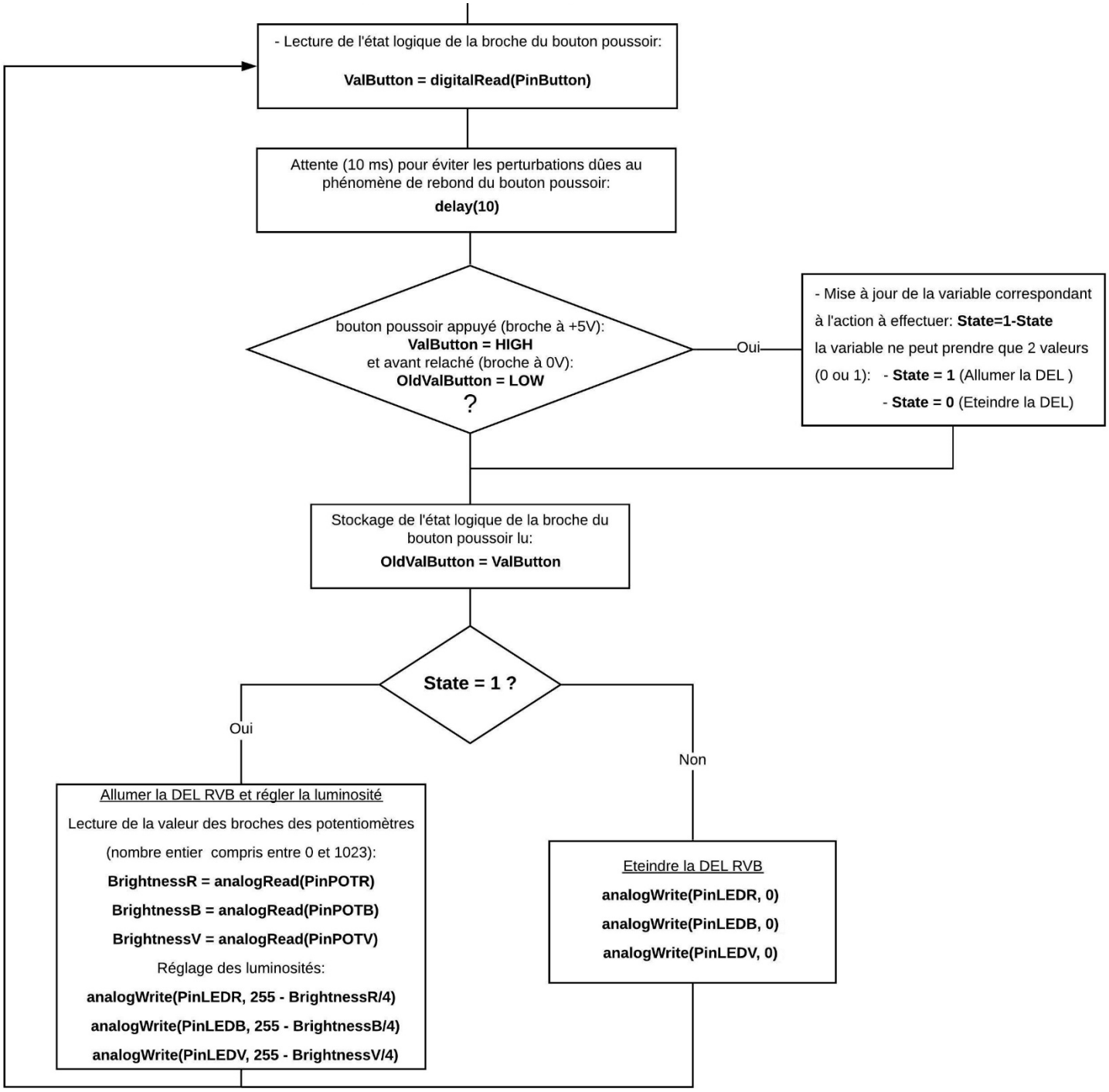
La fenêtre "Synthèse additive des couleurs " est fermée en cliquant sur : 

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

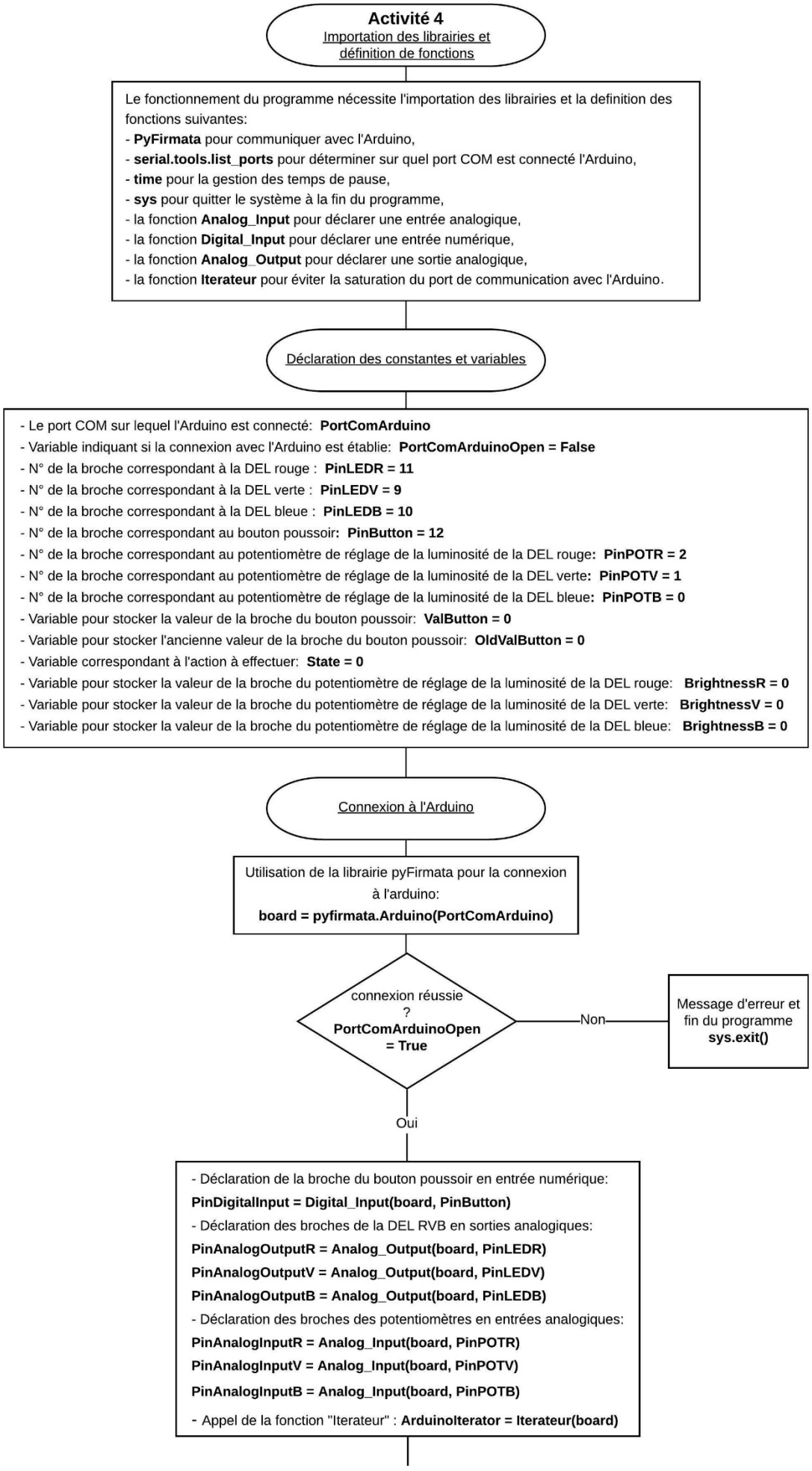
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l’activité sont semblables, comme vous pouvez le voir ci-dessous :

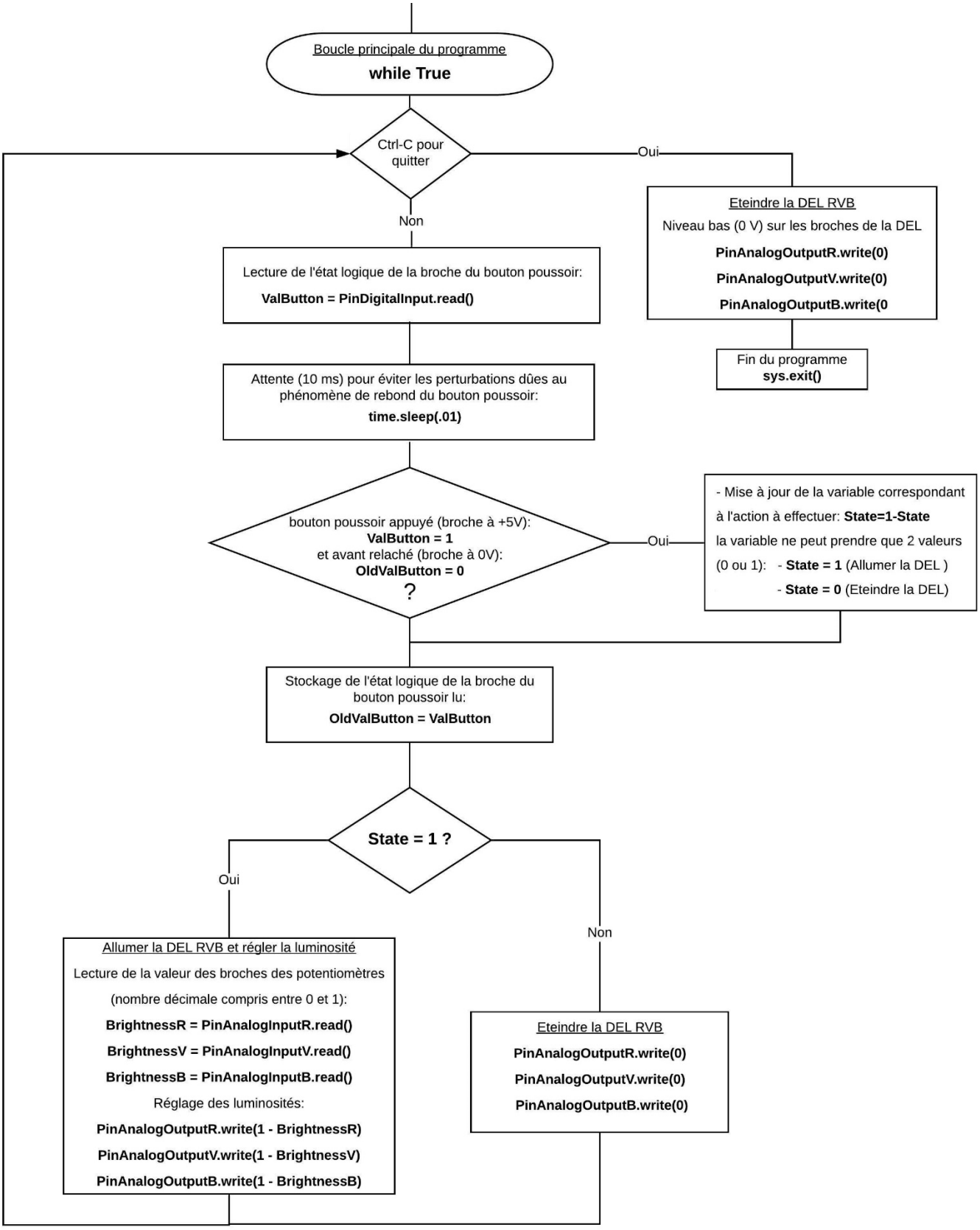
* Algorithme de programmation de l'activité 4 en langage Arduino IDE :




- Lecture de l'état logique de la broche du bouton poussoir:
ValButton = digitalRead(PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = HIGH
et avant relaché (broche à 0V):
OldValButton = LOW
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: **State=1-State**
la variable ne peut prendre que 2 valeurs (0 ou 1): - **State = 1** (Allumer la DEL)
- **State = 0** (Eteindre la DEL)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Allumer la DEL RVB et régler la luminosité
Lecture de la valeur des broches des potentiomètres (nombre entier compris entre 0 et 1023):
BrightnessR = analogRead(PinPOTR)
BrightnessB = analogRead(PinPOTB)
BrightnessV = analogRead(PinPOTV)
Réglage des luminosités:
analogWrite(PinLEDR, 255 - BrightnessR/4)
analogWrite(PinLEDB, 255 - BrightnessB/4)
analogWrite(PinLEDV, 255 - BrightnessV/4)
Eteindre la DEL RVB
analogWrite(PinLEDR, 0)
analogWrite(PinLEDB, 0)
analogWrite(PinLEDV, 0)

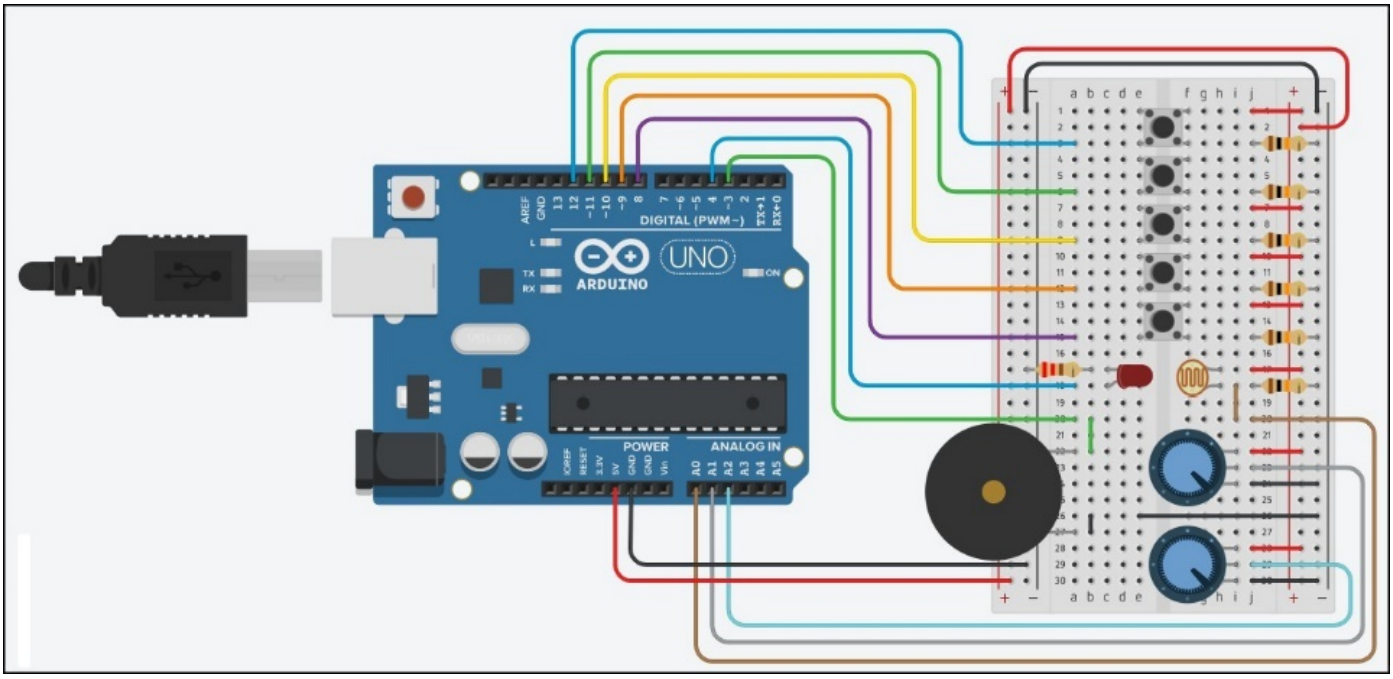
* Algorithme de programmation de l'activité 4 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre la DEL RVB
Niveau bas (0 V) sur les broches de la DEL
PinAnalogOutputR.write(0)
PinAnalogOutputV.write(0)
PinAnalogOutputB.write(0
Fin du programme
sys.exit()
Lecture de l'état logique de la broche du bouton poussoir:
ValButton = PinDigitalInput.read()
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(.01)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1): - State = 1 (Allumer la DEL)
- State = 0 (Eteindre la DEL)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Allumer la DEL RVB et régler la luminosité
Lecture de la valeur des broches des potentiomètres (nombre décimale compris entre 0 et 1):
BrightnessR = PinAnalogInputR.read()
BrightnessV = PinAnalogInputV.read()
BrightnessB = PinAnalogInputB.read()
Réglage des luminosités:
PinAnalogOutputR.write(1 - BrightnessR)
PinAnalogOutputV.write(1 - BrightnessV)
PinAnalogOutputB.write(1 - BrightnessB)
Eteindre la DEL RVB
PinAnalogOutputR.write(0)
PinAnalogOutputV.write(0)
PinAnalogOutputB.write(0)

2.5.3 Ondes sonores : Produire & Exploiter

Après avoir bien travaillé, maintenant que nous maitrisons le principe des entrées et sorties de l'Arduino, nous allons nous détendre en écoutant un peu de musique…



- Liste des composants :

. 1 DEL Rouge
. 1 résistances de 220 Ω
. 6 résistances de 10 kΩ
. 2 potentiomètres de 10 kΩ
. 1 photorésistance
. 6 boutons poussoir
. 1 haut-parleur (ou piezzo)
. 1 plaque d'essai
. Fils de connexion

- Protocole de communication (Mode "Contrôle de l'Arduino") :

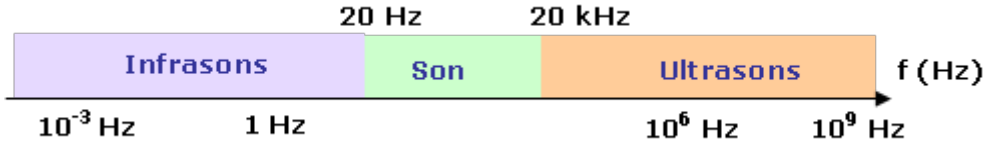
. Firmata Express

Rappels :

Une onde sonore ou acoustique est une perturbation mécanique périodique qui se propage dans un milieu matériel fluide élastique (l'eau ou l'air par exemple), en s'éloignant de sa source. C'est une onde de compression-dilatation du milieu dans lequel elle se propage.

Par exemple, Une corde de guitare qui vibre, va transmettre sa vibration à l'air. Cette vibration va se propager dans l'air jusqu'à nos oreilles qui vont recevoir la vibration.

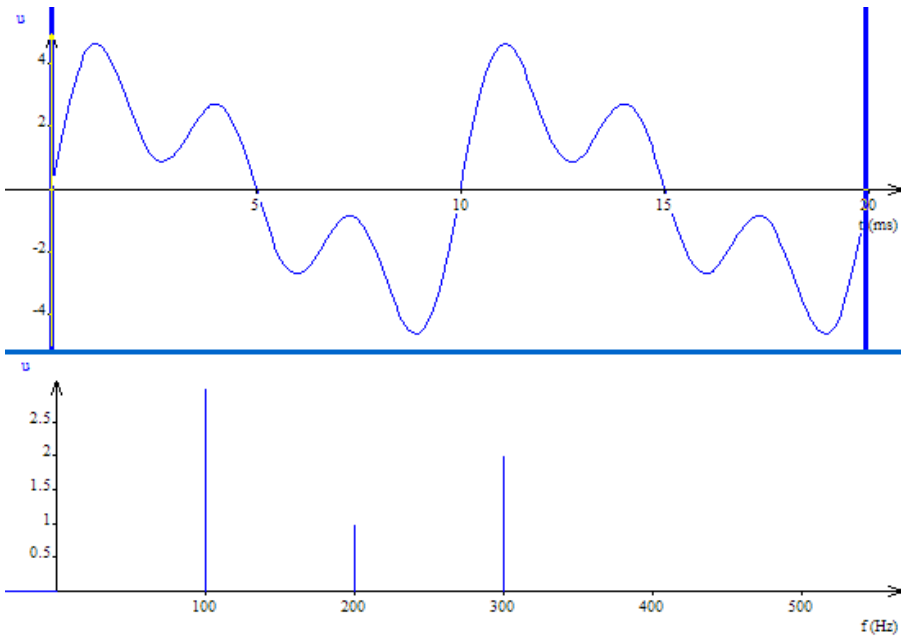
L'être humain peut entendre des sons dont les fréquences s'étalent de 20Hz à 20kHz environ.



Un son pur est un signal sonore sinusoïdale de fréquence **f** (Hz) qui ne varie pas au cours du temps. Le signal sonore délivré par un diapason est un son pur.

La plupart des sons que nous percevons dans notre environnement ne sont pas purs mais complexes. Ils sont composés de plusieurs sons purs de fréquences et d'amplitudes différentes. Un son musical est un cas particulier de son complexe, produit par un instrument de musique. Un son musical est périodique (de période constante au cours du temps), mais n'est pas forcément sinusoïdal. C'est en fait, une superposition de plusieurs signaux sinusoïdaux.

II est possible de décomposer un signal sonore u(t) de fréquence f associé à la propagation d'une onde sonore périodique non sinusoïdale, en une somme infinie de signaux sinusoïdaux, c'est la décomposition de Fourier du signal :



Le signal ci-dessus de fréquence f=100 Hz (T=10 ms) se décompose de la façon suivante :

$$u(t)= 3\times\sin(2\times\pi\times100\times t) +\sin(2\times\pi\times200\times t) + 2\times\sin(2\times\pi\times300\times t)$$

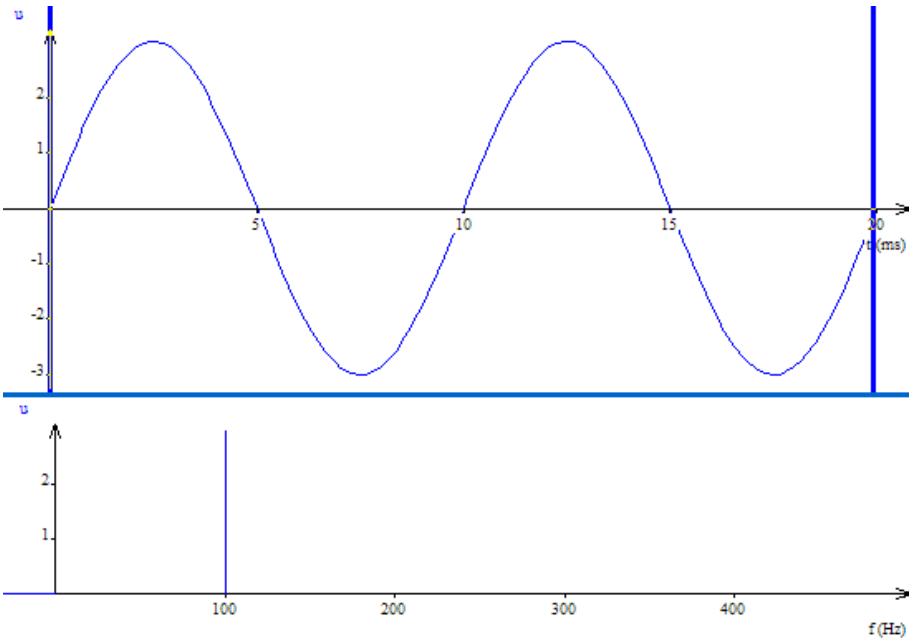
Un signal sonore complexe périodique de fréquence f est donc une superposition de signaux sinusoïdaux :

- un signal sinusoïdal à la fréquence f nommée "fondamental" ou "première harmonique",
- un signal sinusoïdal à la fréquence 2f, la "deuxième harmonique",
- un signal sinusoïdal à la fréquence 3f, la "troisième harmonique", etc…

La représentation de l'amplitude des harmoniques en fonction de la fréquence constitue le spectre du signal (voir image ci-dessus).

Les harmoniques sont des signaux sinusoïdaux de fréquences $f_n=n\times f$. Le nombre ***n*** est un entier positif appelé rang de l'harmonique.

Dans le cas d'un son pur, le spectre du signal sonore ne présente qu'une unique harmonique, le fondamental.



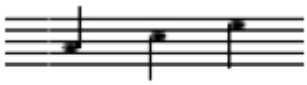
Spectre d'un son pur de fréquence f=100 Hz

Un son musical est caractérisé par sa hauteur et son timbre :

. **<u>Hauteur d'un son musical</u>**

La hauteur d'un son musical est la fréquence **f**, exprimée en hertz (symbole : Hz), de l'onde sonore périodique. C'est la fréquence du fondamental dans la décomposition de Fourier de cette onde.

La hauteur d'un son musical est associée au nom de la note jouée (do, ré, mi, ... d'une certaine octave). Le terme de hauteur vient du fait que les notes sont écrites sur une portée musicale de telle manière que la position verticale de la note sur la portée corresponde à son nom.



La portée musicale

Si la fréquence est multipliée par deux, on passe à l'octave supérieure. À l'inverse si la fréquence est divisée par deux, on passe à l'octave inférieure.

Une note de musique correspond à une fréquence d'un son à toutes les octaves.

La note "**la**" correspond à la fréquence f=440 Hz, mais aussi à 880 Hz (octave supérieure), 220 Hz (octave inférieure), etc.

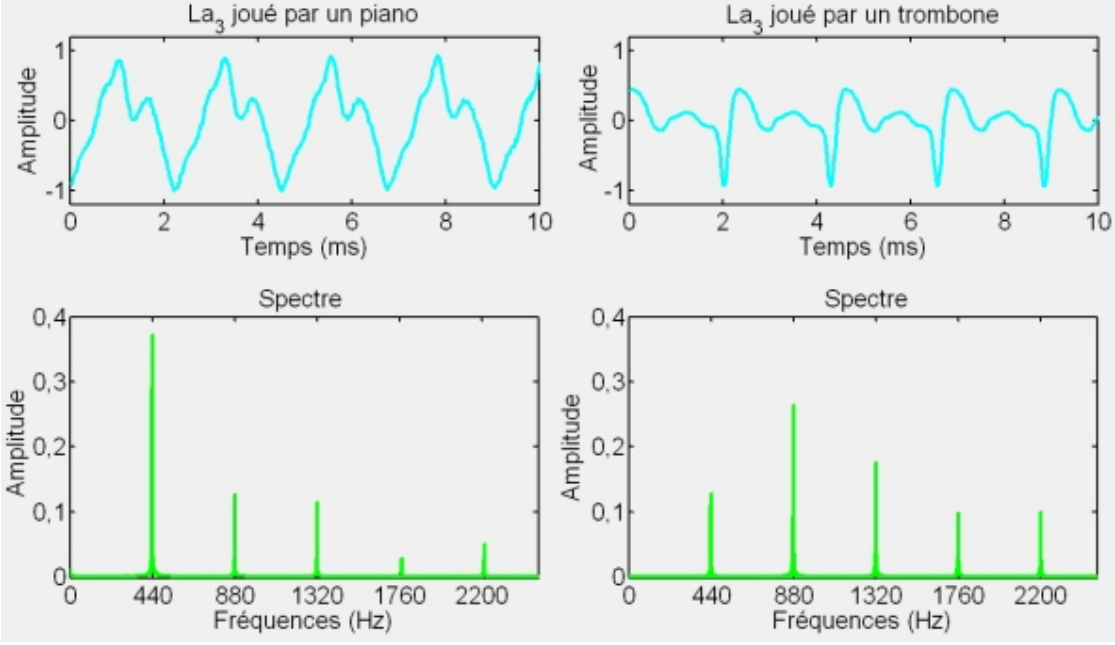
Une onde sonore est d'autant plus aigüe que sa fréquence est grande. Elle est d'autant plus grave que sa fréquence est petite.

. Timbre d'un son musical

Une note de hauteur donnée n'est pas perçue de la même manière selon qu'elle jouée par un diapason ou par un piano. Le timbre du son est différent.

Des sons de même hauteur peuvent donner des sensations différentes en raison de leur timbre. Le timbre d'un son est lié à sa composition spectrale (présence, importance et durée des harmoniques) et à son évolution au cours du temps.

Le timbre d'un son dépend donc de l'instrument qui émet ce son. Les oscillogrammes de deux sons musicaux de même hauteur nous permettent de vérifier que ces deux sons ont même période, donc même fréquence fondamentale. Par contre, les allures de ces deux sons en fonction du temps sont très différentes. Les spectres de Fourier des deux sons font apparaître des harmoniques de mêmes fréquences, mais d'amplitudes différentes d'un son à l'autre.



. **Les différentes notes de musique**

Chaque note est caractérisée par une fréquence fondamentale déterminée. Lorsque deux notes sont séparées d'une octave, le rapport de leur fréquence est égal à deux.

Dans la pratique, ces deux notes ont le même nom, comme par exemple le "**la**" de l'octave **3** et le "**la**" de l'octave **4**, nommée couramment "**la3**" et "**la4**" pour éviter toute ambigüité entre elles.

On appelle gamme l'ensemble des notes (de Do à Si) composant une octave donnée. Dans la gamme tempérée, c'est-à-dire celle utilisée dans la musique occidentale, l'octave est divisée en 12 demi-tons, ce qui correspond à 12 notes, en comptant les notes diésées (#).

Par exemple, pour l'octave 1, voici les valeurs de fréquence des 12 notes:

Note	Fréquence (Hz)
Do	65
Do# ou Réb	69
Ré	74
Ré# ou Mib	78
Mi	83
Fa	87
Fa# ou Solb	93
Sol	98

Note	Fréquence (Hz)
Sol# ou Lab	104
La	110
La# ou Sib	117
Si	123

Le # signifie dièse et le b signifie bémol. Ce sont des altérations des notes de la gamme de base (Do, Ré, Mi, Fa, Sol, La, Si). Le dièse augmente la fréquence de la note et le bémol la diminue. Ainsi un La # est situé en fréquence entre le La et le Si

• En notation latine (la notation historique : do ré mi …)

On a l'habitude d'écrire que le diapason (la note qui nous sert de référence à 440 Hz) est le "**la3**". Dans ce système, on peut se donner en repère que la première octave entièrement audible commence par le do0.

• En notation américaine (la notation scientifique : A B C …)

Dans cette notation, le "**la**" du diapason est le **A4**. C'est le système qui est utilisé en langage Arduino. Dans ce système, on a choisi de dire que le C (do) le plus grave d'un clavier de piano à 88 touches est le C1. Suivant ce repère, on a alors **A4 = 440 Hz**.

Voici un tableau qui référence la fréquence des notes de musique en hertz de la gamme tempérée (notation américaine en noir et notation latine en rouge)

Note \octave	0 (-1)	1 (0)	2 (1)	3 (2)	4 (3)	5 (4)	6 (5)	7 (6)	8 (7)
C (Do)	16,35	32,70	65,41	130,81	261,63	523,25	1046,50	2093,00	4186,01
C♯ – Db (Do♯)	17,32	34,65	69,30	138,59	277,18	554,37	1108,73	2217,46	4434,92
D (Ré)	18,35	36,71	73,42	146,83	293,66	587,33	1174,66	2349,32	4698,64
D♯ – Eb (Ré♯)	19,45	38,89	77,78	155,56	311,13	622,25	1244,51	2489,02	4978,03
E (Mi)	20,60	41,20	82,41	164,81	329,63	659,26	1318,51	2637,02	5274,04
F (Fa)	21,83	43,65	87,31	174,61	349,23	698,46	1396,91	2793,83	5587,65
F♯ – Gb (Fa♯)	23,12	46,25	92,50	185,00	369,99	739,99	1479,98	2959,96	5919,91
G (Sol)	24,50	49,00	98,00	196,00	392,00	783,99	1567,98	3135,96	6271,93
G♯ – Ab (Sol♯)	25,96	51,91	103,83	207,65	415,30	830,61	1661,22	3322,44	6644,88
A (La)	27,50	55,00	110,00	220,00	440,00	880,00	1760,00	3520,00	7040,00
A♯ – Bb (La♯)	29,14	58,27	116,54	233,08	466,16	932,33	1864,66	3729,31	7458,62
B (Si)	30,87	61,74	123,47	246,94	493,88	987,77	1975,53	3951,07	7902,13

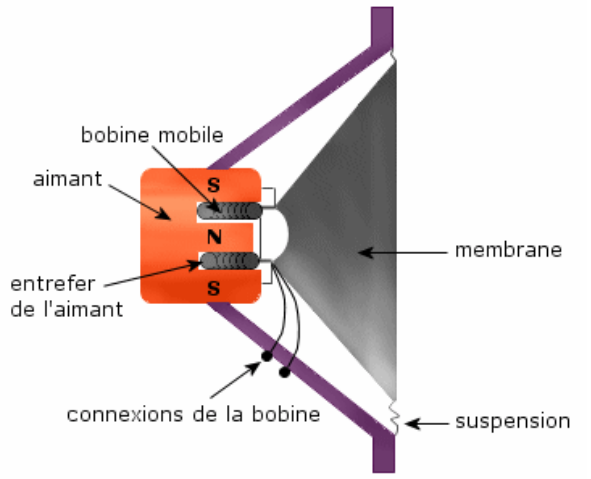
. **Arduino et ondes sonores** :

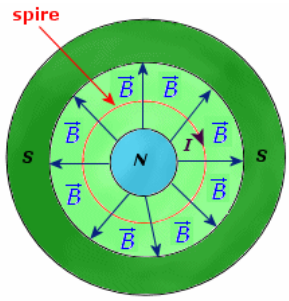
Comme une corde de guitare qui vibre et qui transmet sa vibration à l'air, pour produire un son avec un Arduino, il faut utiliser un matériel qui peut vibrer sur commande !

Pour cela on utilise un petit haut-parleur ou un buzzer (transducteur) piézo-électrique (communément appelé "piezo") connecté sur une des sorties de l'Arduino.

- Principe de fonctionnement du haut-parleur :

Le haut-parleur est constitué principalement :
• d'un **aimant** circulaire ;
• d'une **bobine circulaire** mobile placée autour d'un des pôles de l'aimant ;
• d'une **membrane** reliée à la bobine ;
• d'un **support** qui contient l'aimant, la bobine et la membrane.
Les fils de la bobine sont connectés à la sortie du haut-parleur.





Le champ magnétique créé par l'aimant est perpendiculaire en tout point au courant qui circule dans chacune des spires.

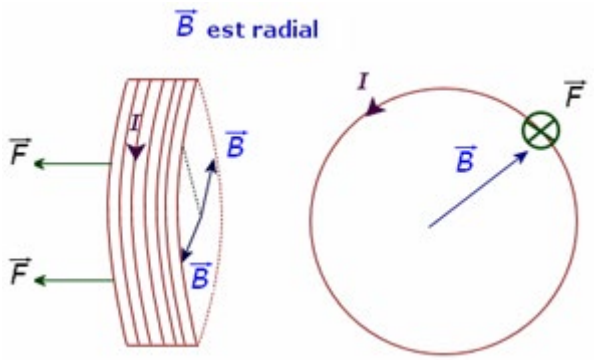
Lorsqu'un courant continu circule dans les spires, il se créé une force appelée force de Laplace. Cette force n'existe que lorsque les spires sont plongées dans un champ magnétique.

La valeur de cette force est proportionnelle à l'intensité du courant I, à la valeur du champ magnétique B et à la longueur L du conducteur qui subit la force.

On écrit : **F = I.L.B** lorsque le champ magnétique et le courant sont perpendiculaires.

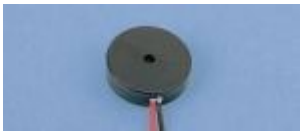
F est exprimé en newton (N), I en ampère (A), L en mètre (m) et B en tesla (T).

La direction et le sens de cette force dépendent de la direction et du sens du champ magnétique et du sens du courant dans les spires.



La sortie de l'Arduino est connectée à la bobine mobile, le courant électrique venant de l'Arduino la traverse, et sous l'action du champ magnétique intense de l'entrefer, va pousser en avant la membrane ou la tirer en arrière suivant la polarité de cette tension électrique. En se déplaçant ainsi d'avant en arrière au rythme du signal électrique, la membrane crée une onde sonore qui se propage dans l'air.

- <u>Principe de fonctionnement d'un buzzer (transducteur) piézo-électrique</u>



Un matériau piézoélectrique est une substance qui produit un courant électrique lorsqu'il est déformé. Et inversement, lorsqu'une tension électrique est appliquée sur la substance, une déformation a lieu.

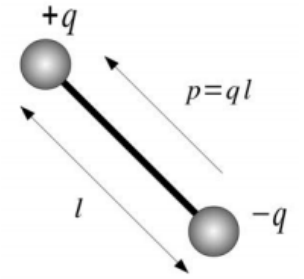
. **Description de la piézoélectricité :**

L'effet piézoélectrique résulte d'un déplacement des atomes (chargés positivement ou négativement) à l'intérieur de certains solides déformables (matériaux piézoélectriques), qui présentent des structures cristallines particulières (on parle de cristal piézoélectrique) ne présentant pas de centre de symétrie.

L'effet piézoélectrique peut être considéré à l'échelle microscopique comme un déplacement interne du barycentre des charges électriques positives et du barycentre des charges électriques négatives dans une même structure cristalline, lorsque tous les

atomes se déplacent les uns par rapport aux autres sous l'effet d'une déformation du cristal.

Lorsque ces barycentres de charges positives et négatives sont distincts, il y a polarisation (électrique) du cristal, qui se traduit par un moment $p = ql$, où q est la charge et l la distance séparant les deux charges (cf figure).



Moment dipolaire

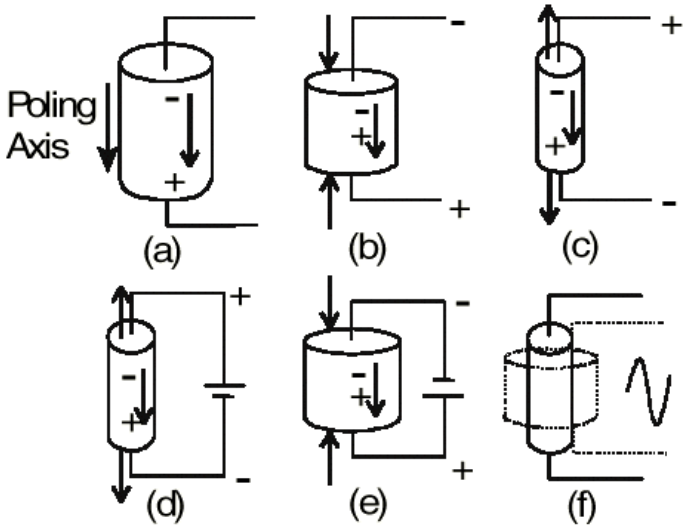
L'effet piézoélectrique peut être considéré à l'échelle macroscopique comme une polarisation électrique d'un solide déformable, sous l'effet de forces appliquées sur sa surface.

Si les faces du solide sont métallisées on peut ramener le problème à un condensateur plan au sein duquel on voit "apparaître" des charges lorsque des forces sont appliquées sur le solide (**figures b et c**)

Réciproquement, si on applique une tension sur les faces du "condensateur", on voit apparaître un champ électrique à l'intérieur du matériau (**figures d, e et f**).

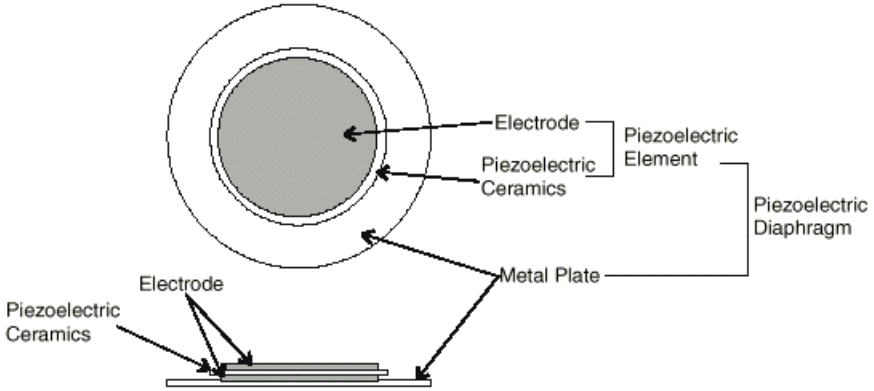
Ce champ sépare les barycentres des charges positives et négatives présentes à l'intérieur du matériau, ce qui peut se traduire soit par une déformation du matériau (si le matériau est libre de se déformer), soit par l'apparition d'une force (si on empêche le matériau de se déformer).

Le schéma ci-dessous illustre l'effet piézo-électrique :



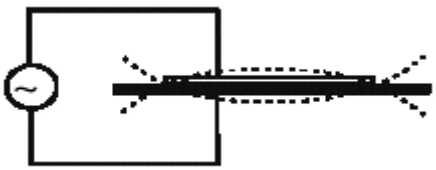
Pour les applications acoustiques, les transducteurs piézoélectriques couramment utilisés sont basés sur le dispositif suivant :

- une plaque piézoélectrique (matériau piézoélectrique entre 2 électrodes métalliques) est collée sur une plaque en laiton, formant ainsi un bilame ;



- lorsqu'un potentiel électrique est appliqué sur les électrodes portées par chaque face du matériau piézoélectrique, celui-ci s'étire ou se compresse, ce qui produit la flexion dans un sens ou dans l'autre de la plaque en laiton et donc du bilame :

- En appliquant une tension sinusoïdale sur les électrodes, le bilame se déforme dans un sens ou dans l'autre à la fréquence du signal appliqué ce qui va créer une onde sonore qui se propage dans l'air :



. **La fonction tone()** :

Le signal électrique appliqué par l'Arduino sur une de ses sorties digitales ou analogiques, sur laquelle est connecté le piezo ou le haut-parleur et qui va créer l'onde sonore, est réalisé avec la fonction **tone()**.

Cette fonction génère une onde carrée (onde symétrique avec "duty cycle" (niveau haut/période) à 50%) à la fréquence spécifiée (en Hertz (Hz) sur une broche. La durée peut être précisée, sinon l'impulsion continue jusqu'à l'appel de l'instruction **noTone()**.
Une seule note peut être produite à la fois. Si une note est déjà jouée sur une autre broche, l'appel de la fonction **tone()** n'aura aucun effet (tant qu'une instruction **noTone()** n'aura pas eu lieu).
Si la note est jouée sur la même broche, l'appel de la fonction **tone()** modifiera la fréquence jouée sur cette broche.
Enfin, l'utilisation de **tone()** rend impossible l'utilisation des broches **D3** et **D11** en PWM avec **analogWrite()**.

SYNTAXE

```
tone(broche, fréquence)
tone(broche, fréquence, durée)
```

PARAMÈTRES

- **Broche** : la broche sur laquelle la note est générée.
- **Fréquence** : la fréquence de la note produite, en hertz (Hz)
- **Durée** : la durée de la note en millisecondes (optionnel)

. **La fonction noTone()** :

La fonction **noTone()** stoppe la génération d'impulsion produite par l'instruction **tone()**. Elle n'a aucun effet si aucune impulsion n'a été générée.

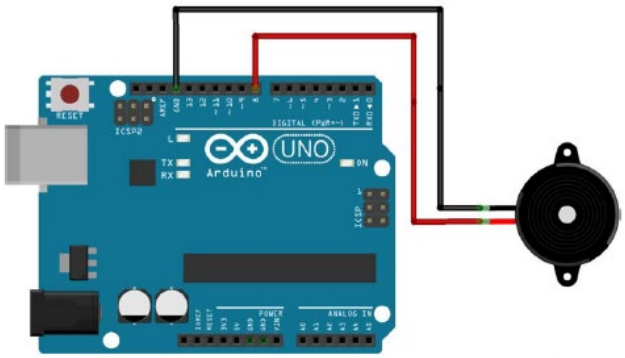
SYNTAXE

```
noTone(broche)
```

PARAMÈTRES

- **broche**: la broche sur laquelle il faut stopper la note.

. <u>Jouer une mélodie avec un Arduino</u> :



On utilisera la fonction Tone() pour jouer les notes de musique (voir tableau des fréquences des notes) de la mélodie pendant la durée définie pour chaque note.

La durée des notes est généralement calculée en attribuant une seconde (1000 ms) à une ronde. Une blanche représente alors 500 ms (ronde/2), une noire, 250 ms (blanche/2, ou ronde /4), une croche,125 ms (noire/2 ou ronde/8), etc…

Pour bien distinguer les notes jouées par l'Arduino, il faut respecter un temps de pause entre chaque note, égal à la durée de la note + 30 % :

Temps de pause (en ms) = Durée de la note (en ms) x 1,3

Pour cela, le plus simple est de créer une liste des fréquences (Freq) des notes à jouer et une liste des durées (Dur) puis de demander à l'Arduino de jouer chaque élément, i, des listes :

tone (broche du piezo, Freq[i], Dur[i])

delay(Dur[i] x 1.3)

- Activité 1 : Faire clignoter une DEL et produire un "beep" synchrone

Dans cette activité, nous allons voir qu'il est possible d'émettre un son avec un Arduino et modifier le premier programme qui permet de faire clignoter une DEL, pour commander la production d'un signal sonore ("un beep"), émis par un piezzo ou un petit haut-parleur, synchrone avec le clignotement de la diode.

Après avoir cliqué sur la prise USB, un menu permettant de régler la fréquence d'émission du "beep" et son rapport cyclique (rapport de la durée d'émission et de la durée de silence) est affiché.

Si le mode de fonctionnement est le "contrôle de l'Arduino", la DEL du circuit réel et la DEL sur l'écran clignote et un "beep" est émis à la fréquence et au rapport cyclique sélectionnés.

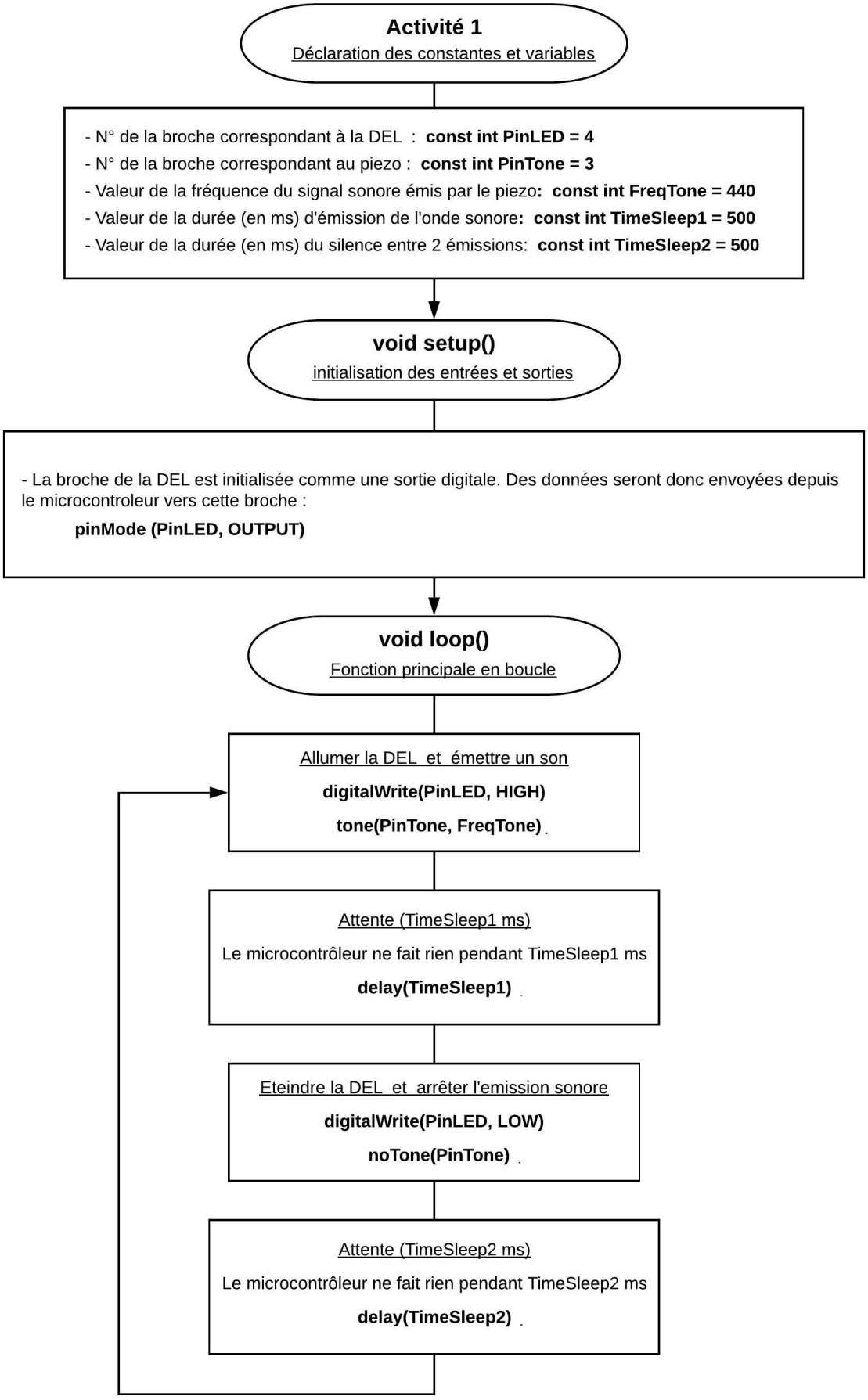
En mode "simulation", **ARDUINO LAB** utilise le lecteur audio du système pour l'émission du signal sonore.
Il est possible que le clignotement de la diode et l'émission sonore soient légèrement désynchronisés.

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

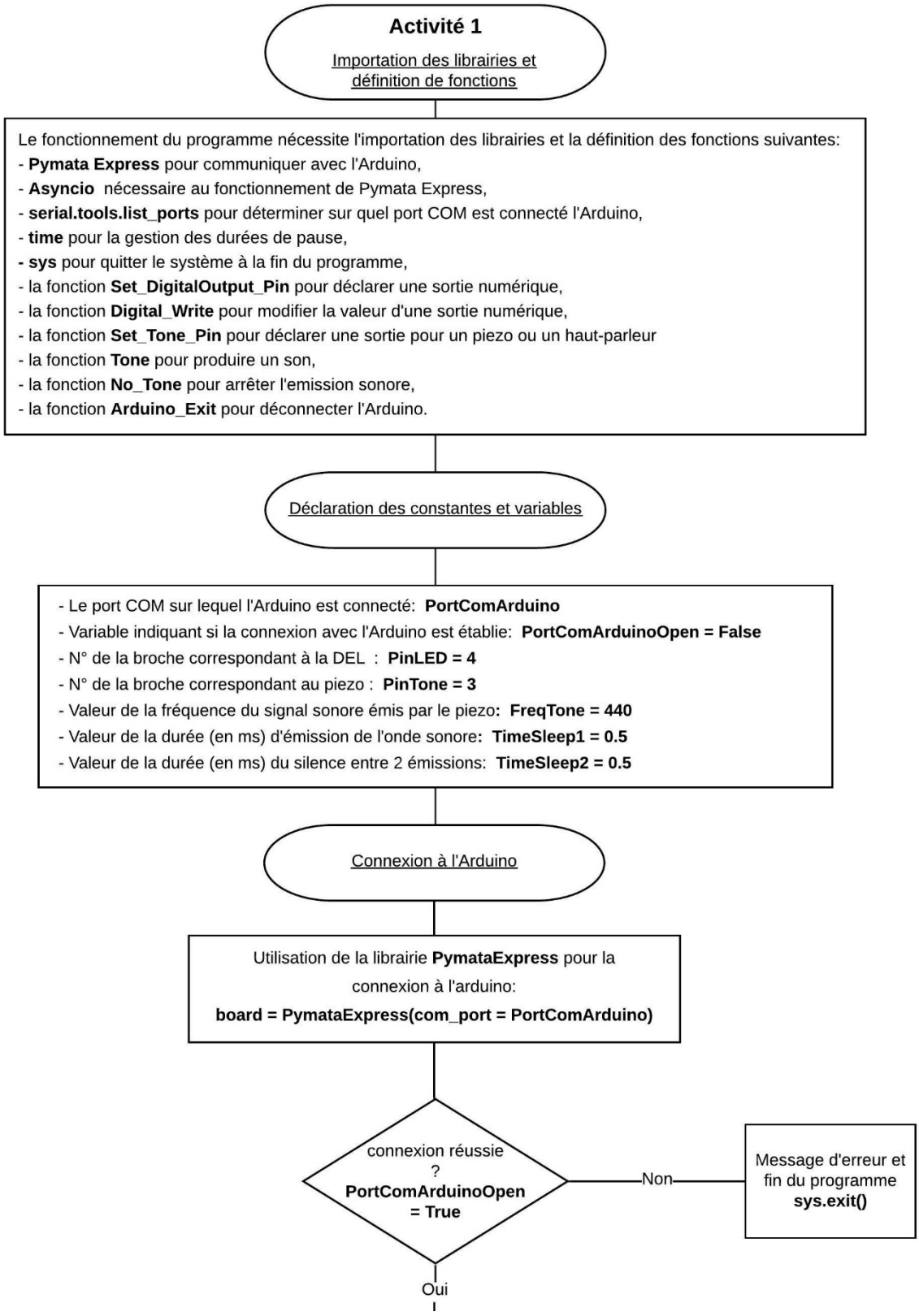
Le code pourra être modifié pour voir l'influence des variables (fréquence de l'onde sonore, durée d'émission, durée de silence).

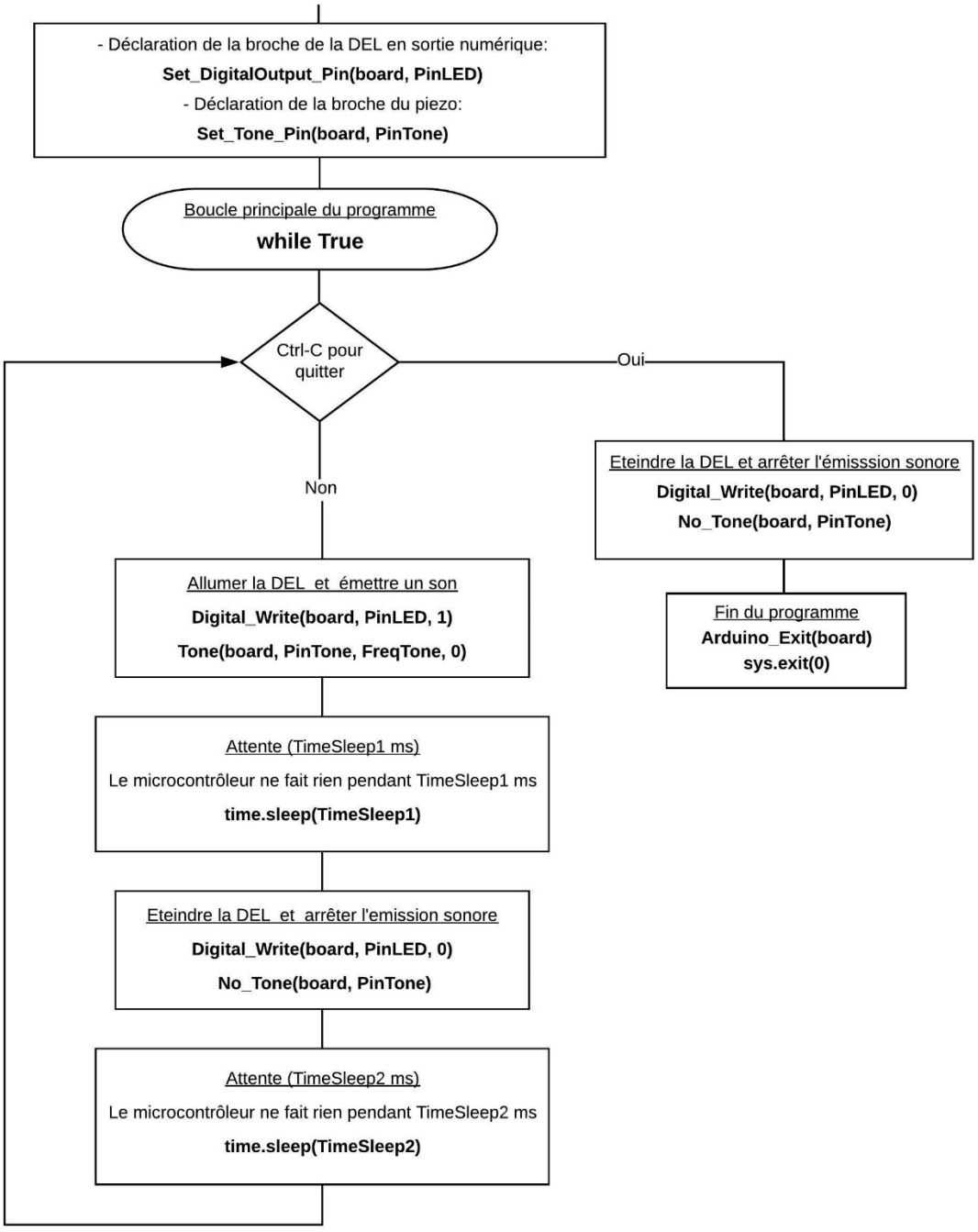
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 1 en langage Arduino IDE :



* Algorithme de programmation de l'activité 1 en Python :




- Déclaration de la broche de la DEL en sortie numérique:
Set_DigitalOutput_Pin(board, PinLED)
- Déclaration de la broche du piezo:
Set_Tone_Pin(board, PinTone)
Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre la DEL et arrêter l'émisssion sonore
Digital_Write(board, PinLED, 0)
No_Tone(board, PinTone)
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Allumer la DEL et émettre un son
Digital_Write(board, PinLED, 1)
Tone(board, PinTone, FreqTone, 0)
Attente (TimeSleep1 ms)
Le microcontrôleur ne fait rien pendant TimeSleep1 ms
time.sleep(TimeSleep1)
Eteindre la DEL et arrêter l'emission sonore
Digital_Write(board, PinLED, 0)
No_Tone(board, PinTone)
Attente (TimeSleep2 ms)
Le microcontrôleur ne fait rien pendant TimeSleep2 ms
time.sleep(TimeSleep2)

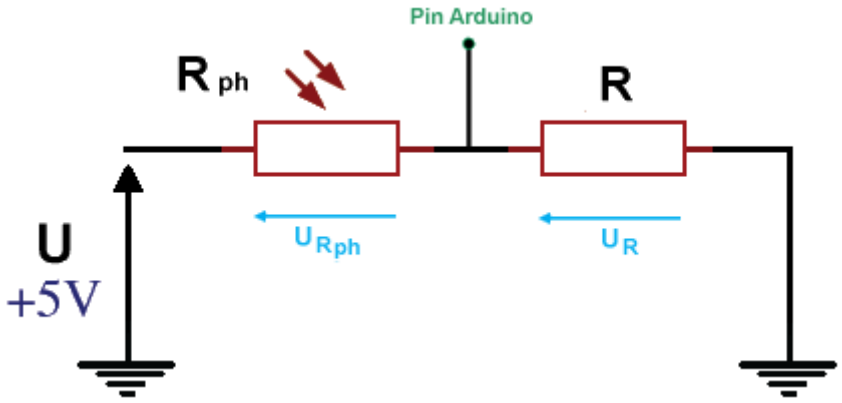
- Activité 2 : Alarme sonore par détection de passage

Dans cette activité, le programme de production d'un "beep" de l'activité précédente va être utilisé comme alarme de détection de passage.

On utilise pour cela une photorésistance éclairée par une DEL rouge. La sortie de la photorésistance est connectée à l'entrée analogique **A0** de l'Arduino.
La valeur de la broche **A0** est alors proportionnelle à l'intensité lumineuse reçue par la photorésistance.

Rappel :

Le circuit avec la photorésistance est représenté ci-dessous :



D'après la loi d'additivité des tensions dans un circuit en série :

$$U = U_{Rph} + U_R = (R_{ph} + R)\, I$$

$$U_R = U - U_{Rph} = U - R_{ph}\, I$$

U_R est la tension appliquée sur l'entrée analogique A0 de l'Arduino. Quand l'intensité lumineuse reçue par la photorésistance augmente, R_{ph} diminue, donc U_R augmente, et au contraire quand la luminosité diminue, R_{ph} augmente et U_R diminue.

On peut exprimer Ur en fonction de U :

$$U = U_{Rph} + U_R = (R_{ph} + R)\, I$$

Donc : $$I = \frac{U}{(R_{ph} + R)}$$

Et : $$U_R = R\, I = \frac{R}{(R_{ph} + R)}\, U$$

En présence d'un obstacle entre la DEL et la photorésistance, la tension mesurée au niveau de la broche A0 diminue et quand celle-ci est inférieure à un seuil (la sensibilité du capteur définie initialement), l'alarme sonore est déclenchée.

Après avoir cliqué sur le connecteur USB, un menu permettant de régler la sensibilité du capteur est affiché (sauf en mode "simulation") et les DELs réelle et virtuelle s'allument.

Si le mode de fonctionnement est le "contrôle de l'Arduino", en présence d'un obstacle entre la DEL et la photorésistance réelles déclenche l'alarme si la sensibilité de déclenchement a été bien réglée.

De même, l'alarme est déclenchée lors du passage de la souris entre la DEL et la photorésistance virtuelles.

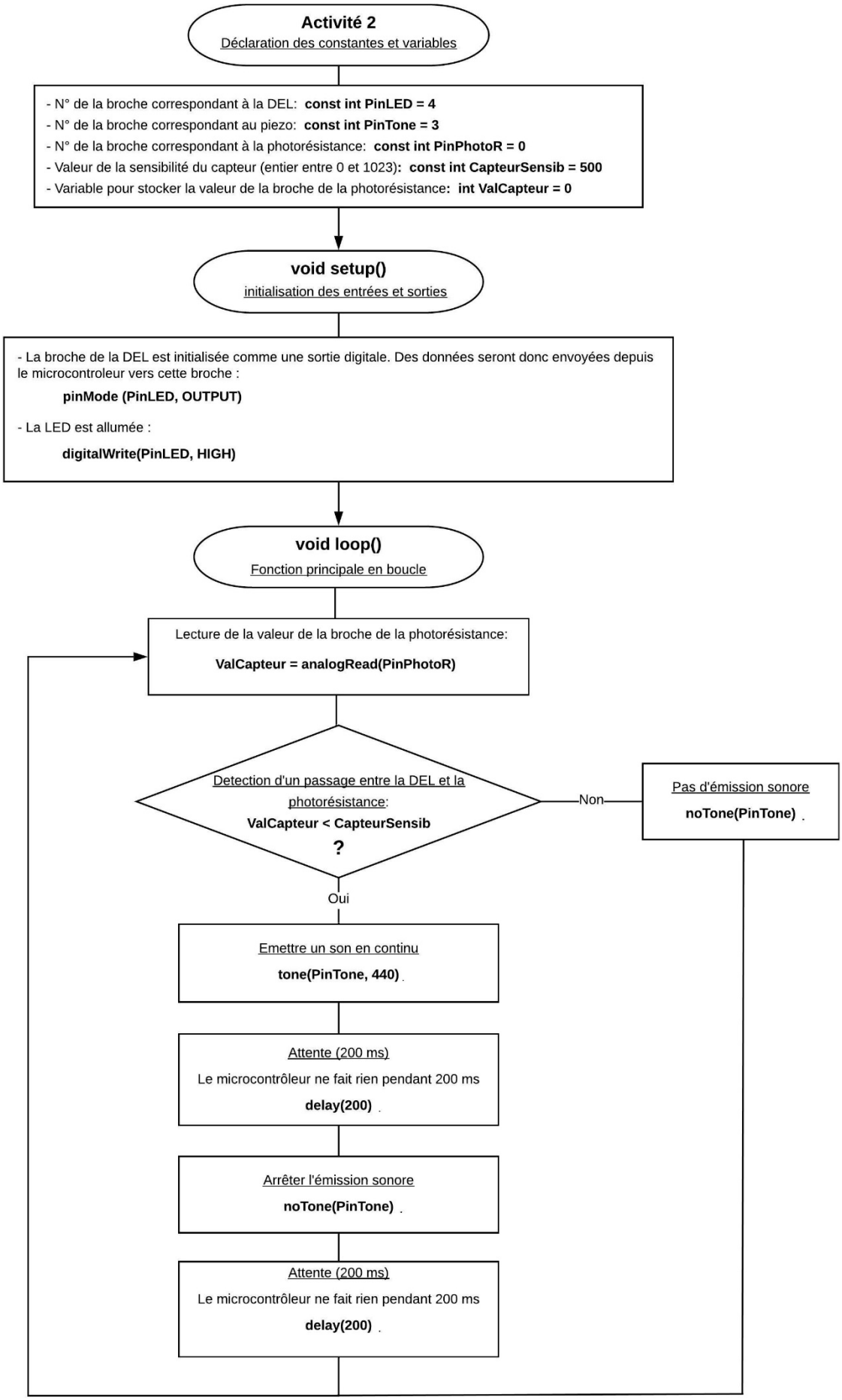
Il s'agit donc bien d'une alarme sonore par détection de passage.

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

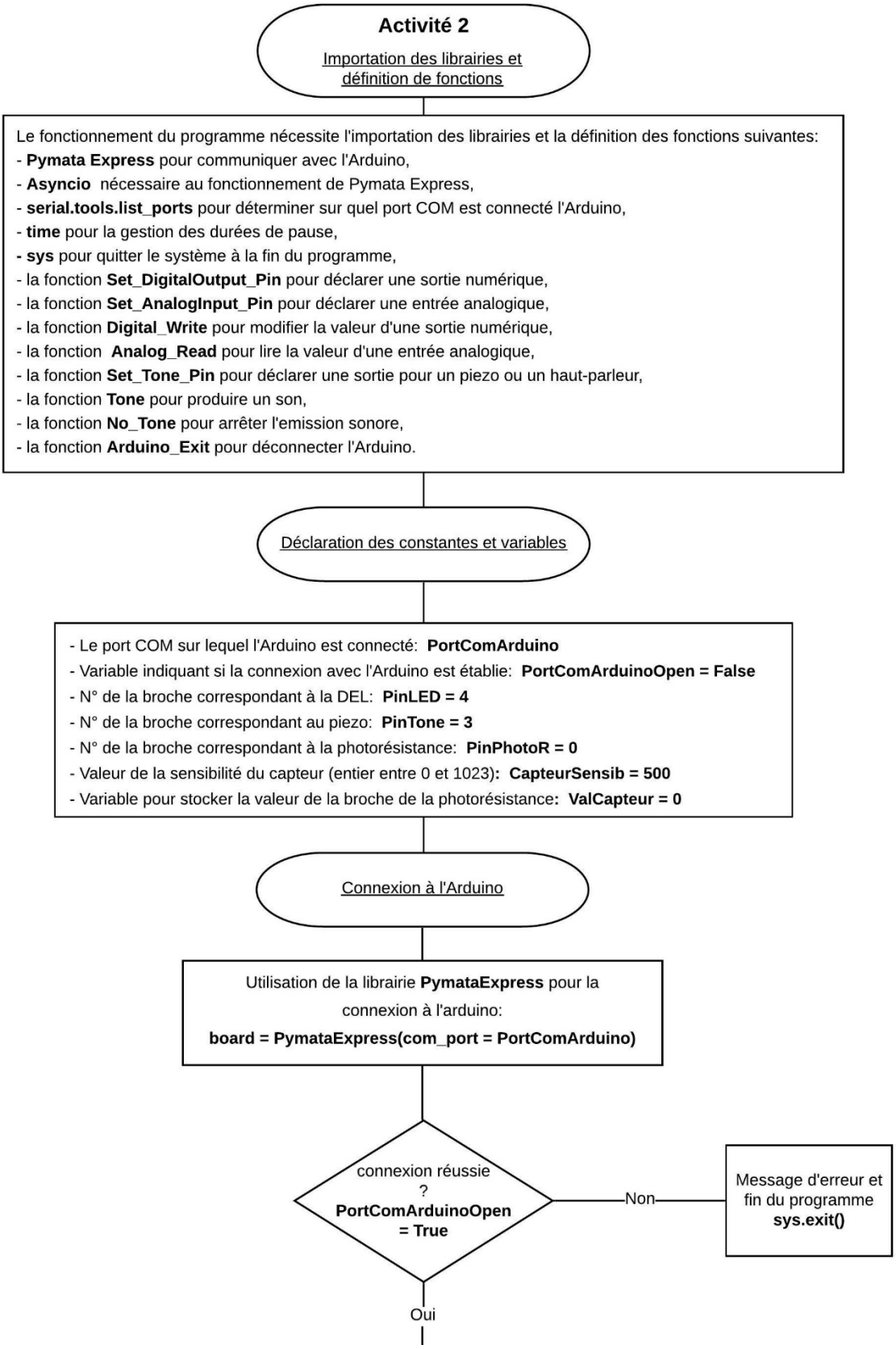
Le code pourra être modifié pour voir l'influence des variables (sensibilité du capteur, fréquence de l'onde sonore, durée d'émission, durée de silence).

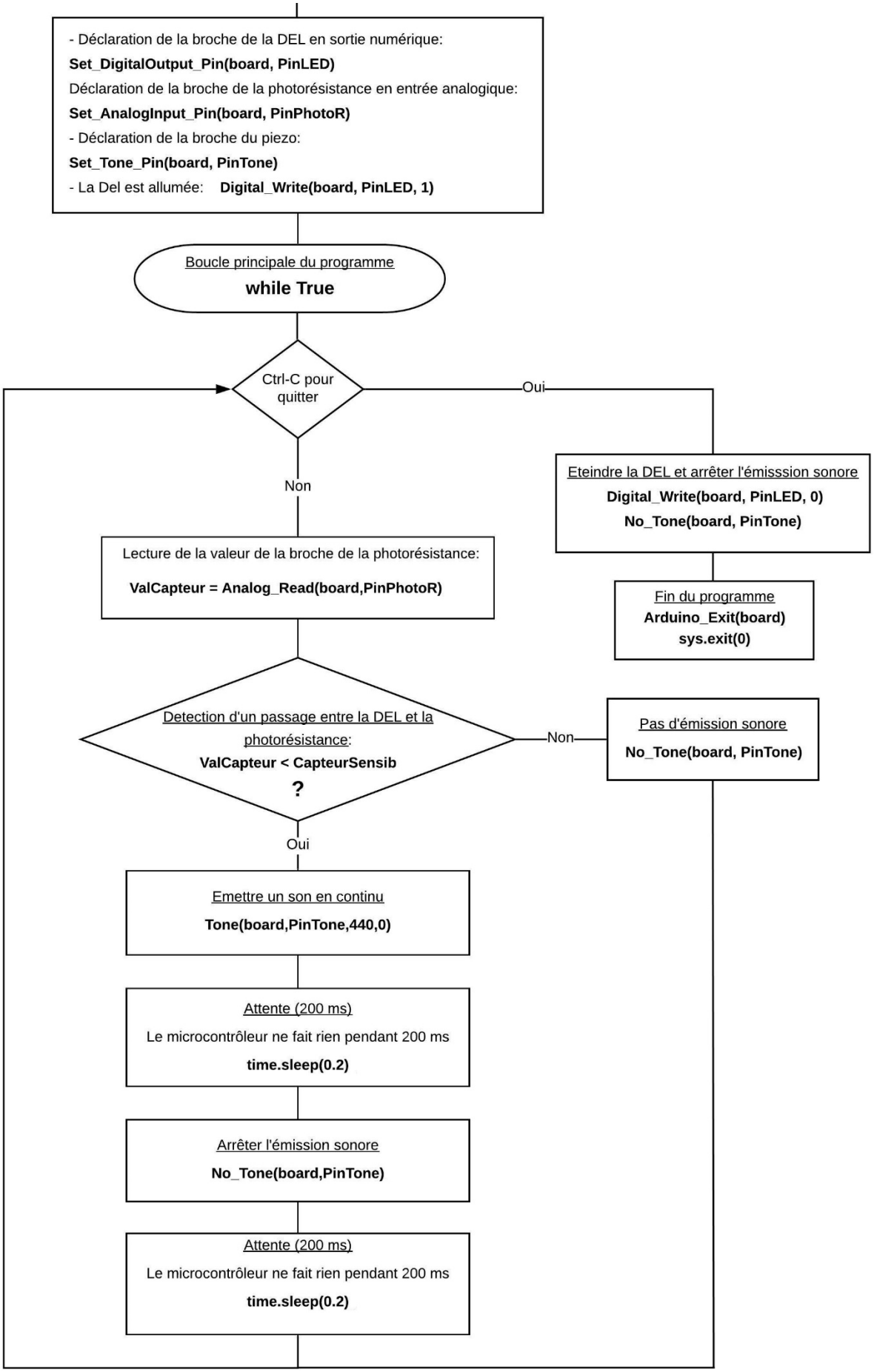
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 2 en langage Arduino IDE :



* Algorithme de programmation de l'activité 2 en Python :




- Déclaration de la broche de la DEL en sortie numérique:
Set_DigitalOutput_Pin(board, PinLED)
Déclaration de la broche de la photorésistance en entrée analogique:
Set_AnalogInput_Pin(board, PinPhotoR)
- Déclaration de la broche du piezo:
Set_Tone_Pin(board, PinTone)
- La Del est allumée: Digital_Write(board, PinLED, 1)
Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Eteindre la DEL et arrêter l'émisssion sonore
Digital_Write(board, PinLED, 0)
No_Tone(board, PinTone)
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Lecture de la valeur de la broche de la photorésistance:
ValCapteur = Analog_Read(board,PinPhotoR)
Detection d'un passage entre la DEL et la photorésistance:
ValCapteur < CapteurSensib
?
Non
Pas d'émission sonore
No_Tone(board, PinTone)
Oui
Emettre un son en continu
Tone(board,PinTone,440,0)
Attente (200 ms)
Le microcontrôleur ne fait rien pendant 200 ms
time.sleep(0.2)
Arrêter l'émission sonore
No_Tone(board,PinTone)
Attente (200 ms)
Le microcontrôleur ne fait rien pendant 200 ms
time.sleep(0.2)

- Activité 3 : Emettre une onde sonore avec un bouton-poussoir

Dans cette activité, nous allons commander la production d'une onde sonore de fréquence préalablement choisie en appuyant sur le premier bouton poussoir. L'émission est arrêtée en relâchant le bouton poussoir.

Après avoir cliqué sur le connecteur USB, un menu permettant de régler la fréquence (en Hz) de l'onde sonore est affiché.

Si le mode de fonctionnement est le "contrôle de l'Arduino", un appui sur le premier bouton poussoir réel ou virtuel déclenche l'émission d'une onde sonore de fréquence égale à la valeur sélectionnée et affichée (de 100 à 1000 Hz par pas de 1 Hz).

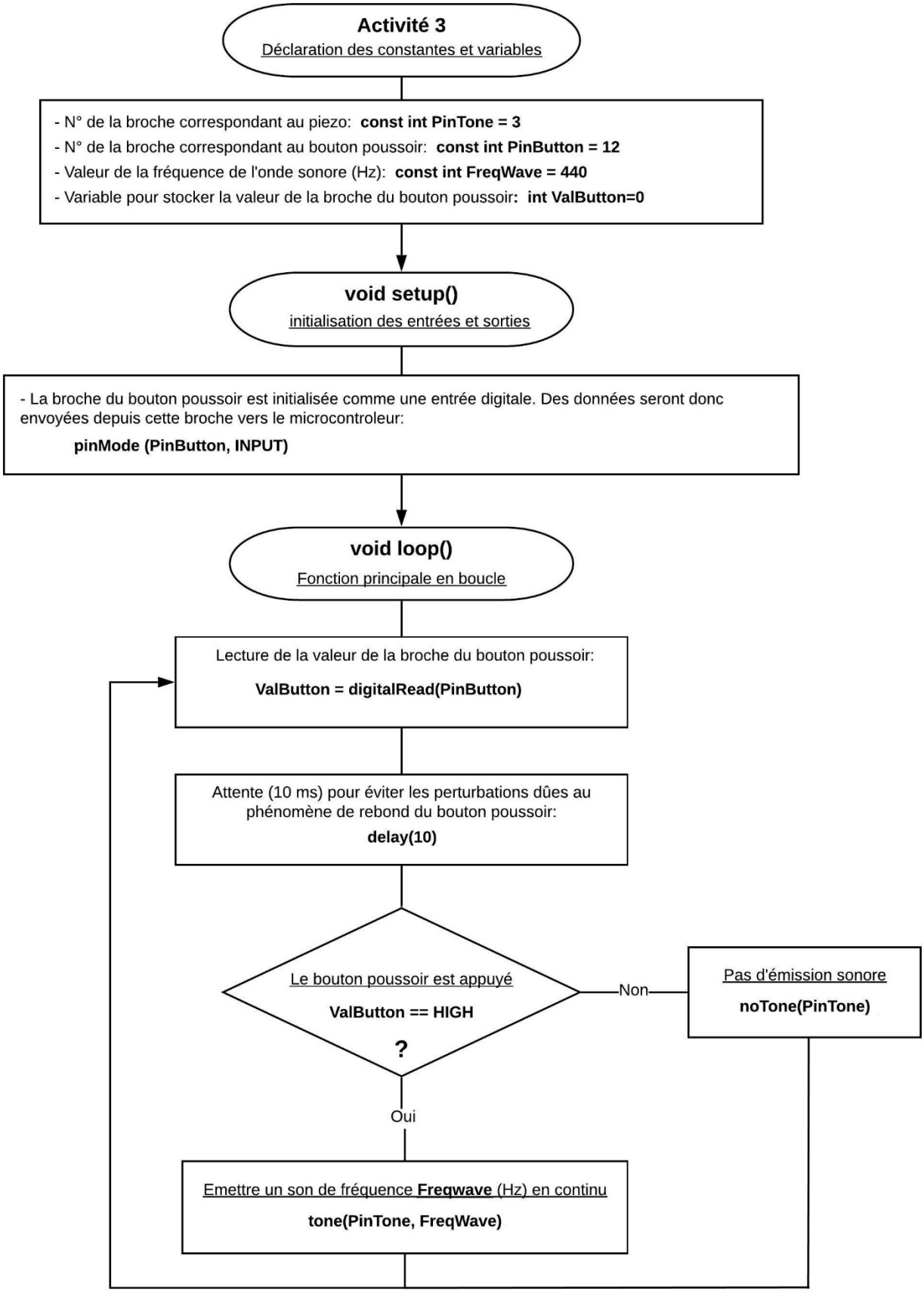
En mode "simulation", ARDUINO LAB utilise le lecteur audio du système pour l'émission du signal sonore. Les fréquences d'ondes disponibles sont celles des notes de musique de A2 (110 Hz) à C8 (4186 Hz).

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

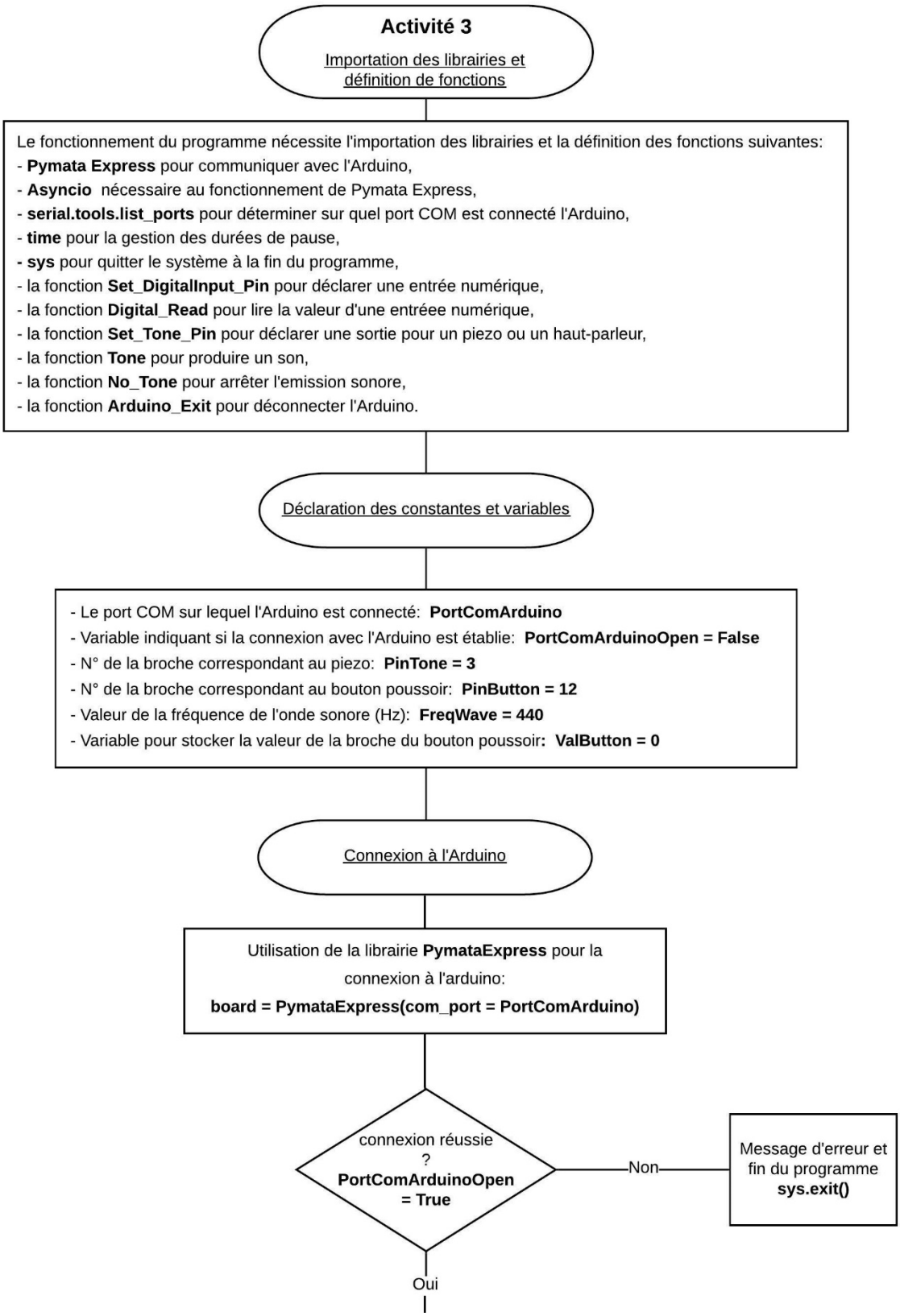
Le code pourra être modifié pour voir l'influence des variables (fréquence onde sonore en Hz).

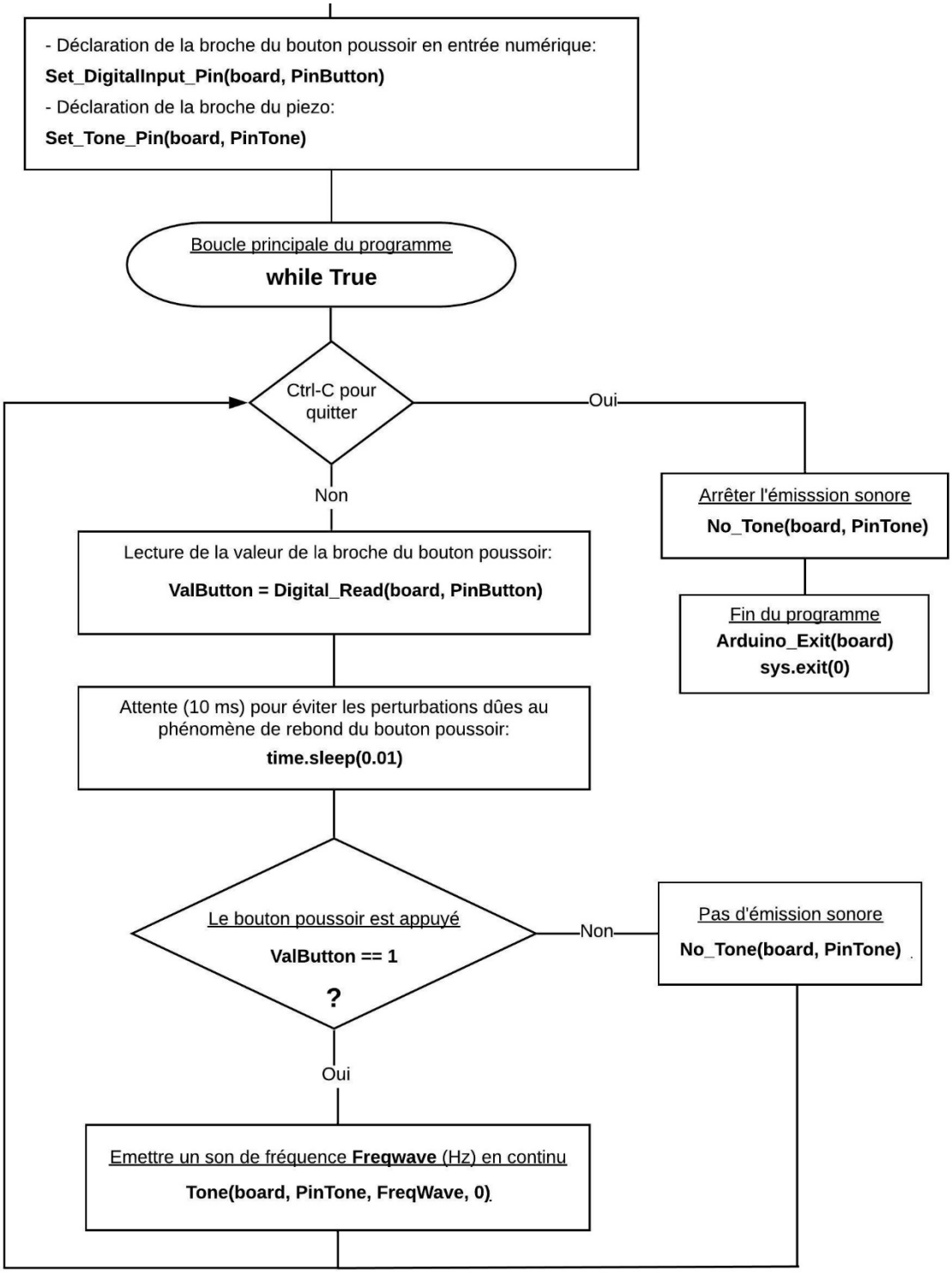
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 3 en langage Arduino IDE :



* Algorithme de programmation de l'activité 3 en Python :




- Déclaration de la broche du bouton poussoir en entrée numérique:
Set_DigitalInput_Pin(board, PinButton)
- Déclaration de la broche du piezo:
Set_Tone_Pin(board, PinTone)
Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Arrêter l'émisssion sonore
No_Tone(board, PinTone)
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Lecture de la valeur de la broche du bouton poussoir:
ValButton = Digital_Read(board, PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(0.01)
Le bouton poussoir est appuyé
ValButton == 1
?
Non
Pas d'émission sonore
No_Tone(board, PinTone)
Oui
Emettre un son de fréquence Freqwave (Hz) en continu
Tone(board, PinTone, FreqWave, 0)

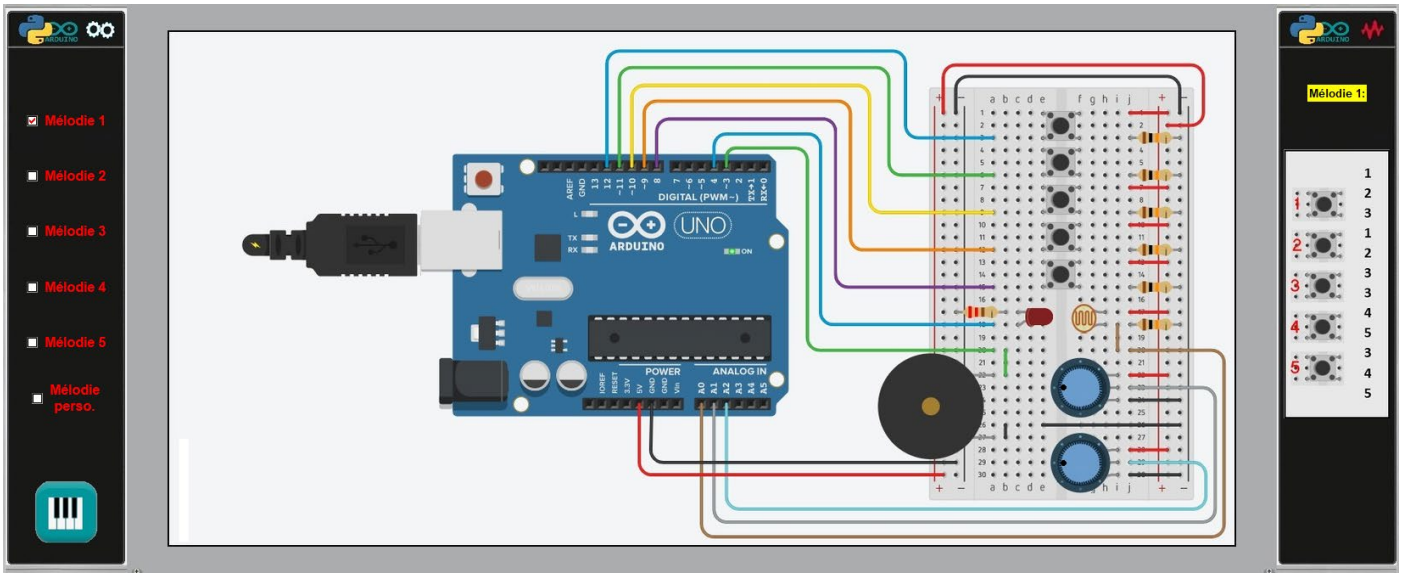
- Activité 4 : Jouer une mélodie avec des boutons poussoir

Dans cette activité, nous allons voir qu'il est possible de jouer une mélodie avec un Arduino et des boutons poussoir qui vont simuler les touches d'un piano :

- On dispose de 5 boutons poussoir que l'on associe chacun à une note de musique (une onde sonore de fréquence déterminée en Hz - voir tableau des fréquences des notes de musique) et à une durée d'émission,

- L'appui sur un bouton poussoir permet de jouer la note associée au bouton pendant la durée définie.

Après avoir cliqué sur le connecteur USB, deux menus permettant de choisir une mélodie et montrant la partition de la mélodie choisie sont affichés de chaque côté du circuit :

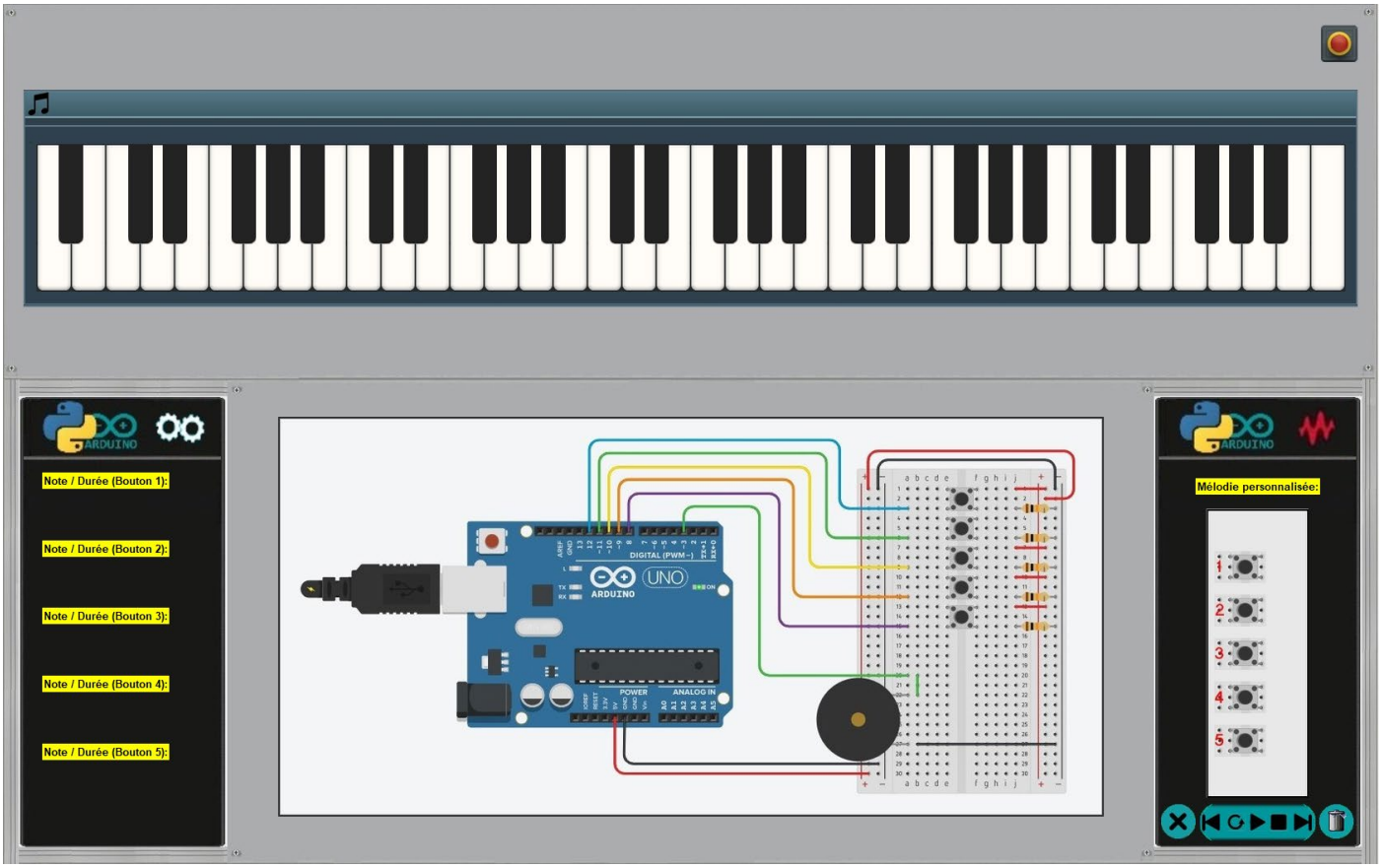


Si le mode de fonctionnement est le "contrôle de l'Arduino", la mélodie est jouée aussi bien en appuyant sur les boutons poussoir réels que virtuels, en suivant la partition (liste dans l'ordre des boutons à appuyer)

En mode "simulation", ARDUINO LAB utilise le lecteur audio du système pour jouer les notes. Le rythme de la mélodie est plus lent.

Vous disposez de 5 mélodies préenregistrées et de la possibilité de créer une mélodie personnalisée en cliquant sur le bouton : 

En cliquant sur ce bouton, une nouvelle fenêtre apparait :



Il est alors possible de jouer une mélodie directement en cliquant sur les touches du piano (notes de musique de A2 à C8) ou de concevoir une mélodie selon ce procédé :

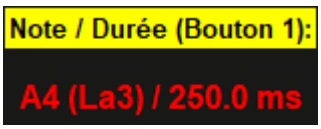
- dans le menu de gauche, cliquer sur un encadré en fond jaune représentant chacun une note associée à un bouton, le fond de l'encadré devient alors rouge,

- ensuite, cliquer sur une touche de piano correspondant à la note souhaitée pour le bouton sélectionné (La note et sa fréquence est affichée quand une touche du piano est survolée avec la souris),

- un encadré représentant différents types de note (ronde, blanche, noire, croche, double croche) apparait alors, cliquer sur le type de note souhaité afin de régler la durée pendant laquelle la note sera jouée,

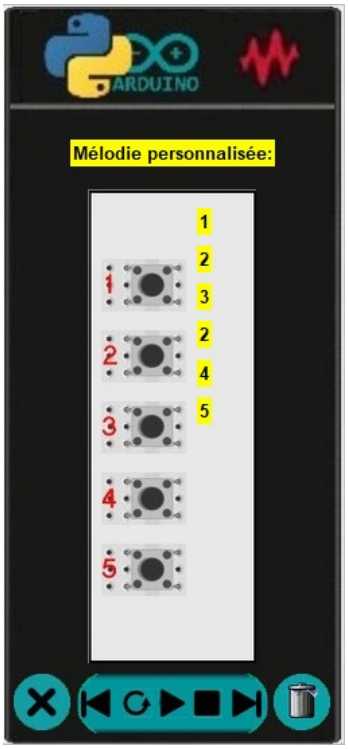
. Ronde = 1000 ms
. Blanche = 500 ms
. Noire = 250 ms
. Croche = 125 ms
. Double croche = 67,5 ms

- La note et sa durée sont alors affichées sous l'encadré (redevenu en fond jaune) précédemment sélectionné :



,

-Dans le menu "Mélodie personnalisée", composer votre mélodie en cliquant sur les boutons du menu. La mélodie est affichée au fur et à mesure,



- La mélodie créée est supprimée en cliquant sur :

 ,

- La dernière note de la mélodie est effacée en cliquant sur :

 ,

- la barre de lecture  permet de jouer la mélodie personnalisée (de façon répétée en option) ou celles préenregistrées en cliquant sur :  ou  pour sélectionner une mélodie.

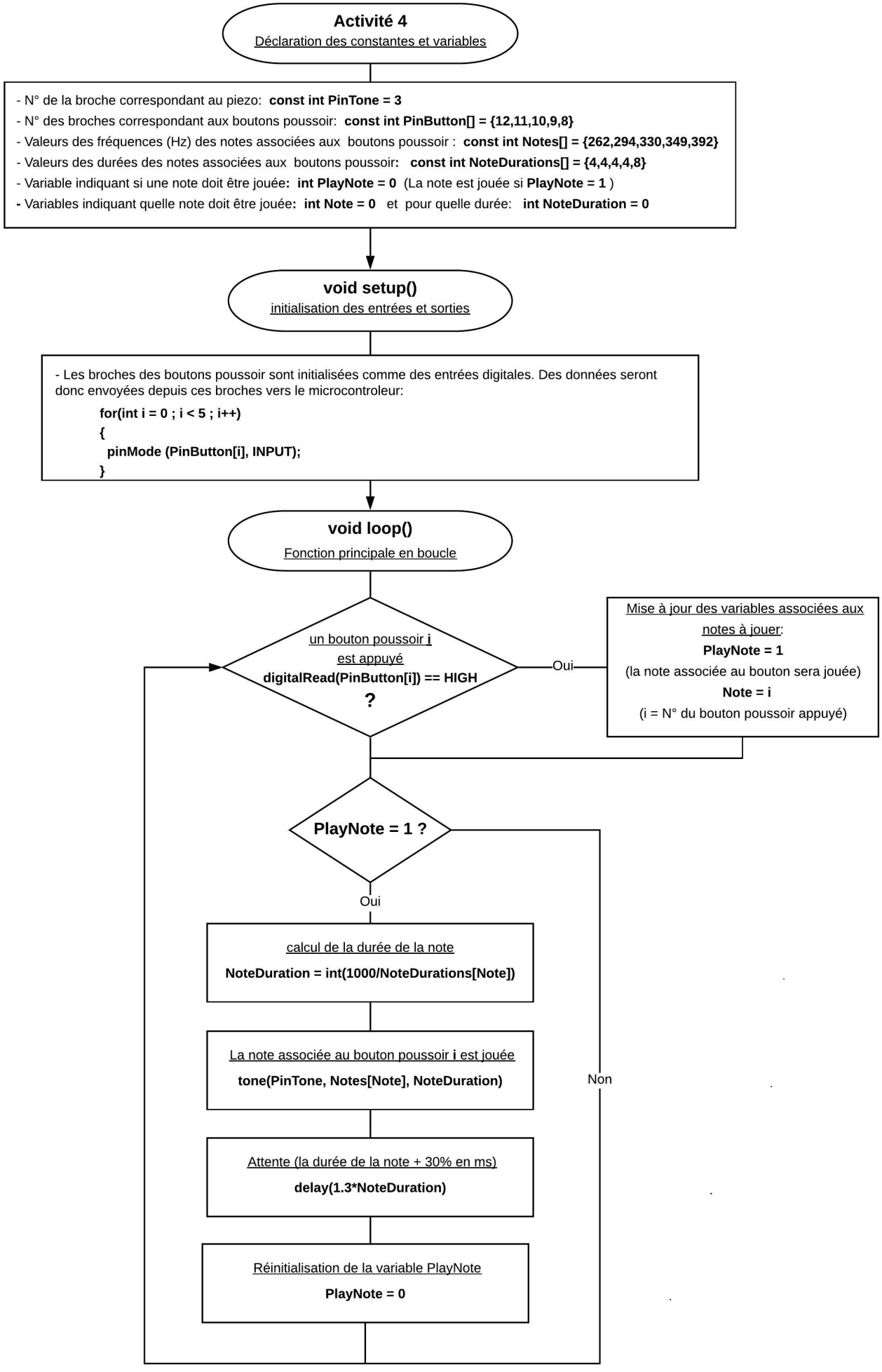
La fenêtre "Mélodie personnalisée" est fermée en cliquant sur : 

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

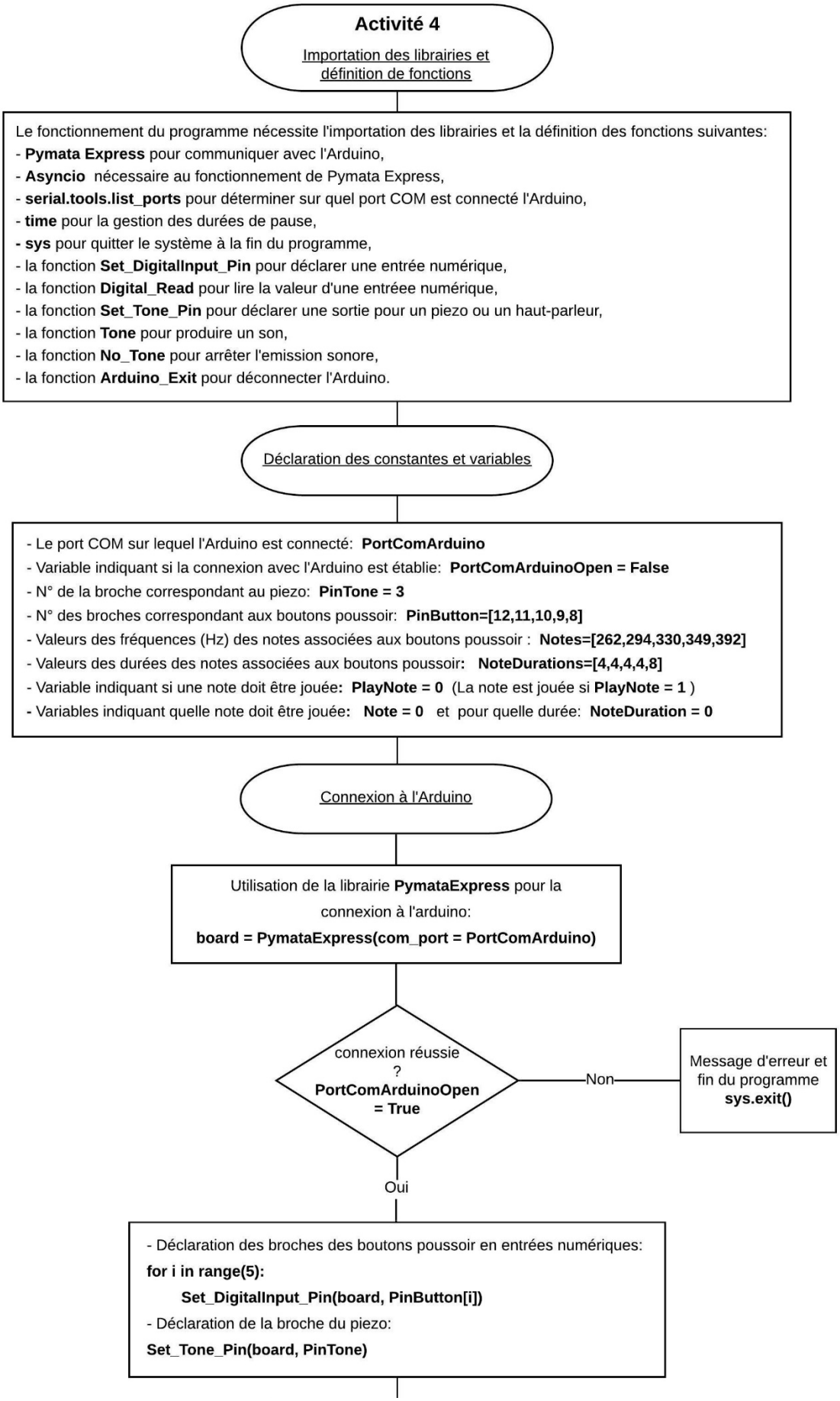
Le code pourra être modifié pour voir l'influence des variables (fréquence des notes associées aux boutons en Hz, durée de la note).

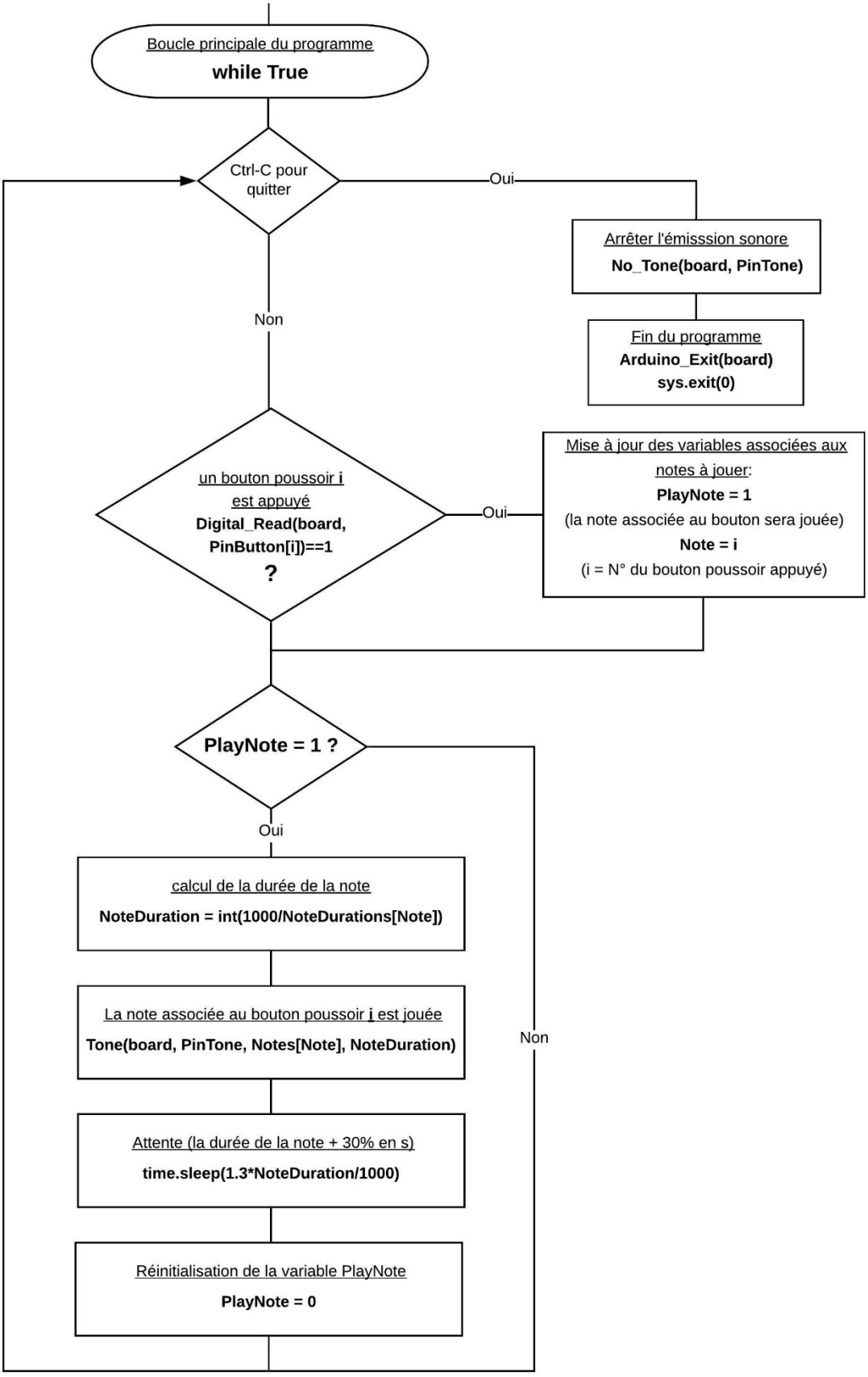
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

* Algorithme de programmation de l'activité 4 en langage Arduino IDE :



* Algorithme de programmation de l'activité 4 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Arrêter l'émisssion sonore
No_Tone(board, PinTone)
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Non
un bouton poussoir i est appuyé
Digital_Read(board, PinButton[i])==1
?
Oui
Mise à jour des variables associées aux notes à jouer:
PlayNote = 1
(la note associée au bouton sera jouée)
Note = i
(i = N° du bouton poussoir appuyé)
PlayNote = 1 ?
Oui
Non
calcul de la durée de la note
NoteDuration = int(1000/NoteDurations[Note])
La note associée au bouton poussoir i est jouée
Tone(board, PinTone, Notes[Note], NoteDuration)
Attente (la durée de la note + 30% en s)
time.sleep(1.3*NoteDuration/1000)
Réinitialisation de la variable PlayNote
PlayNote = 0

- Activité 5 : Régler la fréquence d'une onde sonore avec deux potentiomètres

Dans cette dernière activité, l'appui sur le premier bouton-poussoir produit une onde sonore dont la fréquence est réglée à l'aide de 2 potentiomètres :

- le premier potentiomètre permet un réglage rapide de la fréquence entre 0 et 4080 Hz,
- le deuxième potentiomètre effectue un réglage fin de la fréquence sur une plage de 255 Hz,
- l'émission sonore est arrêtée en appuyant de nouveau sur le bouton poussoir.

Le potentiomètre de réglage rapide est connecté sur la broche **A1** de l'Arduino. La tension de cette broche varie donc entre **0** et **5** V (voir le principe de fonctionnement du potentiomètre) en fonction de la position du curseur du potentiomètre. La lecture de la valeur de la broche **A1** convertie par le convertisseur analogique numérique de l'Arduino donne donc un nombre entier entre **0** et **1023**.

Ce nombre est divisé par 4 de façon à obtenir un nombre entier compris entre **0** et **255** qui sera convertie en nombre binaire (sur 8 bits) :

0 en décimal = **00000000** en binaire
255 en décimal = **11111111** en binaire

Ce nombre binaire sur 8 bits est convertie en nombre binaire sur 12 bits en ajoutant 4 bits de poids faibles, **0000**, à sa fin. On obtient donc un nombre binaire (sur 12 bits) compris entre **000000000000** et **111111110000**, soit en décimal, un nombre entier entre **0** et **4080.**

Le potentiomètre de réglage fin est connecté sur la broche **A2** de l'Arduino. Selon le même principe que précédemment, la lecture de la broche **A2** donne une valeur comprise entre **0** et **1023**.

Ce nombre est également divisé par 4 et convertie en nombre binaire sur 12 bits. On obtient donc un nombre binaire compris entre **0000000000** et **00001111111111** (entre 0 et 255 en décimal).

La conversion en décimal de l'addition des deux nombres binaires (issus de A1 et A2) nous donnent la valeur de la fréquence en Hz de l'onde sonore, soit entre **0** et **4335** Hz avec un pas de réglage de 1 Hz.

Rappel :

En informatique, outre la base 10, on utilise très fréquemment le système binaire (base 2) puisque la logique booléenne est à la base de l'électronique numérique. Deux symboles suffisent : 0 et 1. Cette unité élémentaire ne pouvant prendre que les valeurs 0 et 1 s'appelle un bit (de l'anglais binary digit). Une suite de huit bits s'appelle un octet.

Le tableau ci-dessous montre la représentation des nombres de 0 à 15 dans les bases 10 et 2 :

Décimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binaire	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

. Conversion du décimal en binaire :

La méthode de conversion la plus simple est celle de la division euclidienne par 2. Elle est facile à utiliser en programmation (il est facile d'en faire un algorithme). Voilà comment on fait :

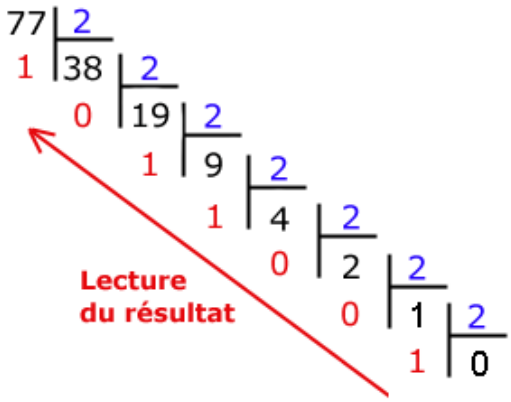
. On a notre nombre en décimal, par exemple : **77**

. On le divise par 2 et on note le reste de la division (c'est soit un 1 soit un 0).

. On refait la même chose avec le quotient précédent, et on met de nouveau le reste de côté.

. On réitère la division, et ce jusqu'à ce que le quotient soit égale à 0.

Le nombre en binaire apparaît : le premier à placer est le dernier reste non nul. Ensuite, on remonte en plaçant les restes que l'on avait. On les place à droite du premier 1 :



77 s'écrit donc en base 2 : **1001101**

. Conversion du binaire en décimal :

Le nombre binaire **1001101** est composé de 7 bits et chaque bit correspond à une puissance de 2. Le premier (en partant de la droite) est le bit de la puissance 0, le deuxième celui de la puissance 1, le troisième celui de la puissance 3, etc…

Pour le convertir un nombre binaire en décimal, on procède de la manière suivante :

On multiplie par 2^0 la valeur du premier bit, par 2^1 la valeur du deuxième bit, par 2^2 la valeur du troisième bit , […], par 2^{10} la valeur du onzième bit, etc… et on fait la somme des résultats :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	1	1	0	1

Le nombre binaire **1001101** en base 10 est :

$2^6 + 2^3 + 2^2 + 2^0 = 64 + 8 + 4 + 1 = 77$

Après avoir cliqué sur le connecteur USB, un menu permettant de visualiser la fréquence (en Hz) de l'onde sonore est affiché.

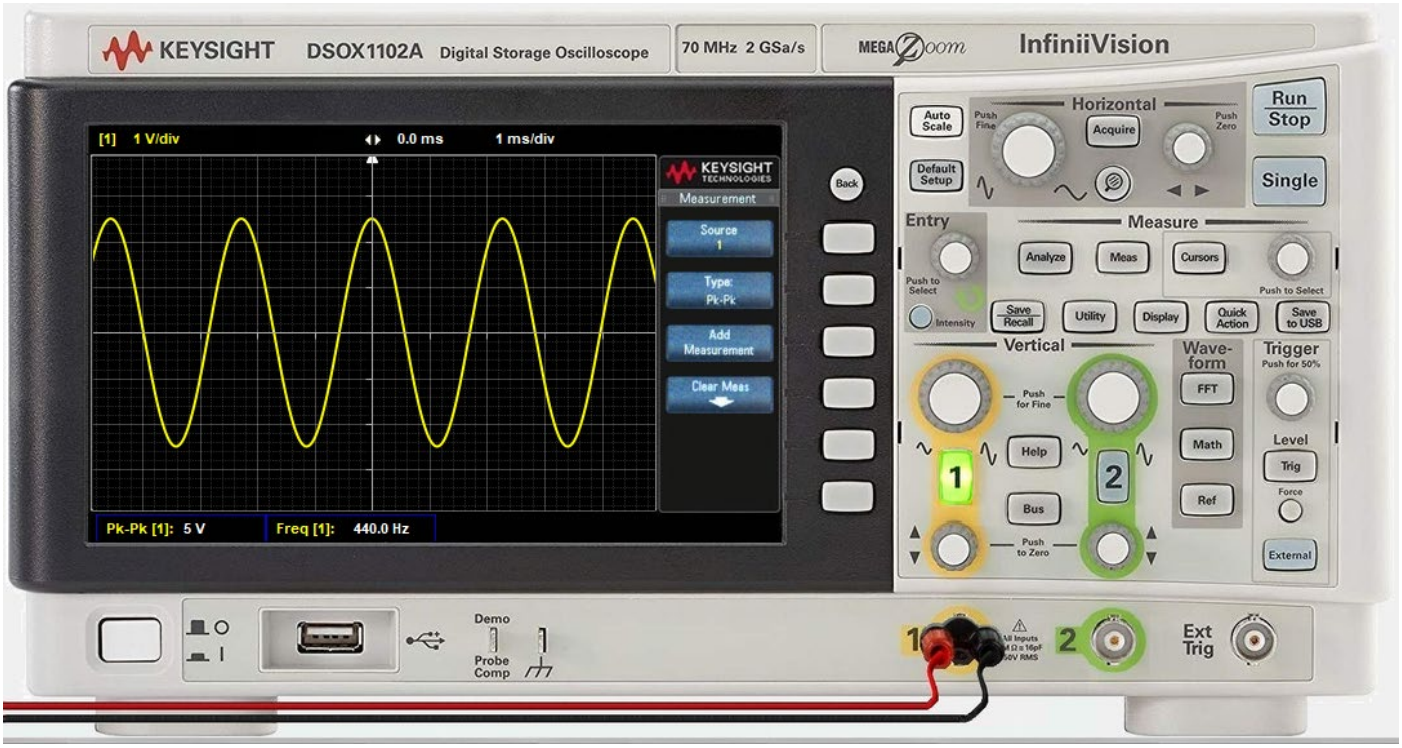
Si le mode de fonctionnement est le "contrôle de l'Arduino", un appui sur le premier bouton poussoir réel ou virtuel déclenche l'émission de l'onde sonore. Le réglage de la fréquence est fait aussi bien avec les potentiomètres réels qu'avec les virtuels (par utilisation de la molette de la souris quand celle-ci est au-dessus du potentiomètre)

En mode "simulation", ARDUINO LAB utilise le lecteur audio du système d'exploitation pour l'émission du signal sonore. Les fréquences d'ondes disponibles sont celles des notes de musique de A2 (110 Hz) à C8 (4186 Hz).

Il est possible de visualiser la simulation de l'observation de l'acquisition de l'onde sonore par un microphone sur un oscilloscope en cliquant sur ce bouton :

Une nouvelle fenêtre est alors affichée :

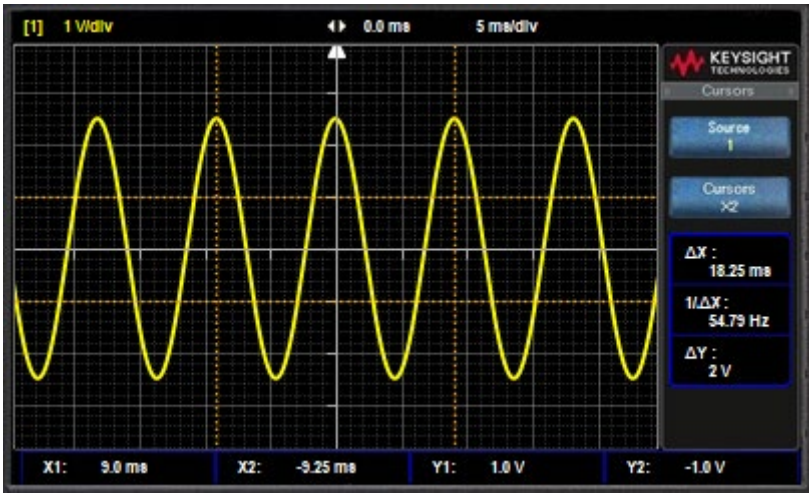
Après avoir cliqué sur l'interrupteur On/Off de l'oscilloscope, l'onde sonore, de fréquence F en Hz, acquise par le microphone est affichée :



Tous les réglages classiques d'un oscilloscope (balayage horizontal, sensibilité verticale, affichage des mesures, curseurs, …) sont simulés et donc modifiables (soit par clic sur les boutons, soit par utilisation de la molette de la souris quand celle-ci est au-dessus d'un bouton de réglage) permettant ainsi l'apprentissage de son utilisation.

Les valeurs d'amplitude et de fréquence sont affichées en appuyant sur : 

ou en utilisant les curseurs en cliquant sur : 



En mode "contrôle de l'Arduino", la modification de la fréquence de l'onde sonore par les potentiomètres est visible sur l'oscilloscope.

Enfin, le haut-parleur se déplace pour visualiser l'influence, sur l'amplitude du signal, de la distance entre la source sonore (le haut-parleur) et le capteur d'acquisition (le microphone).

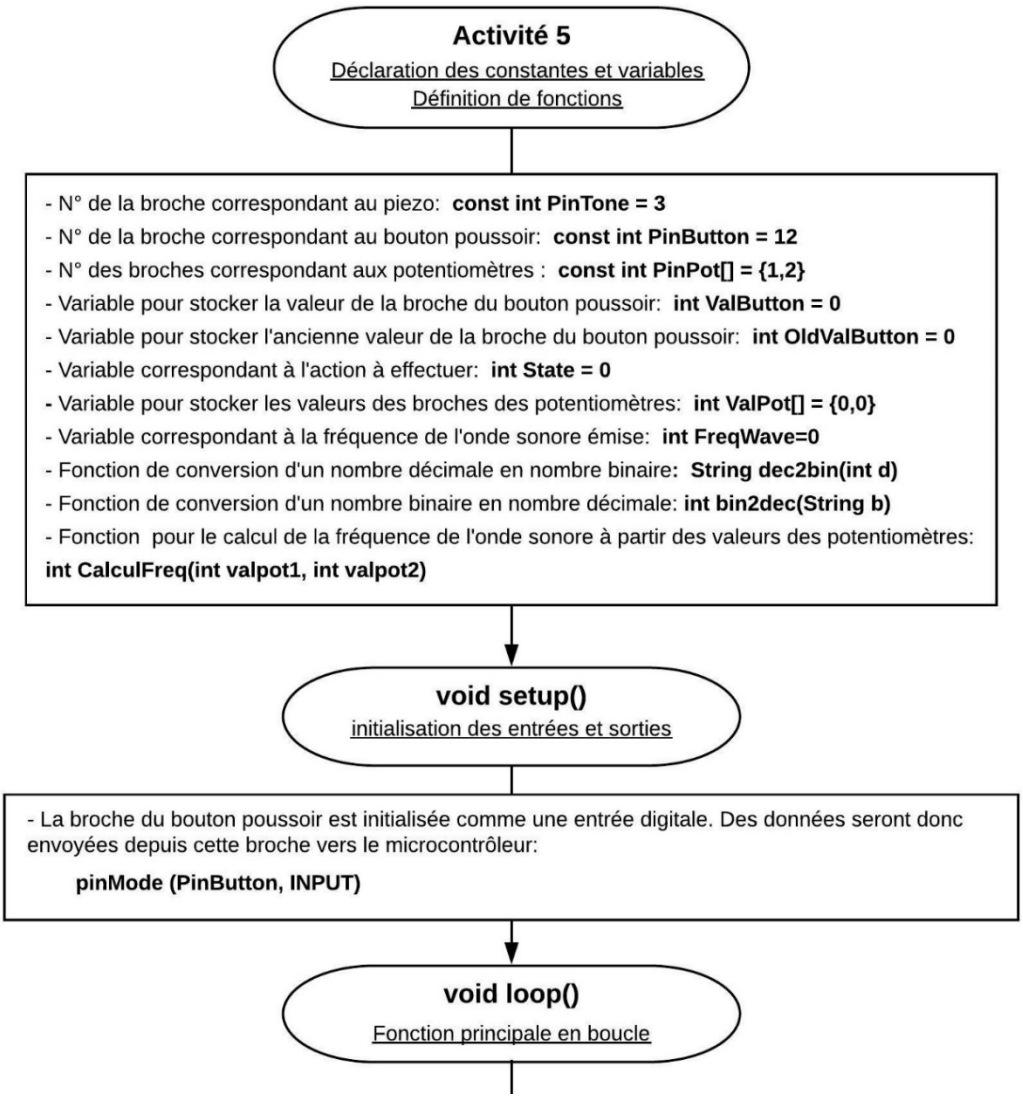
La fenêtre "Acquisition / Observation de l'onde sonore " est fermée en cliquant sur : 

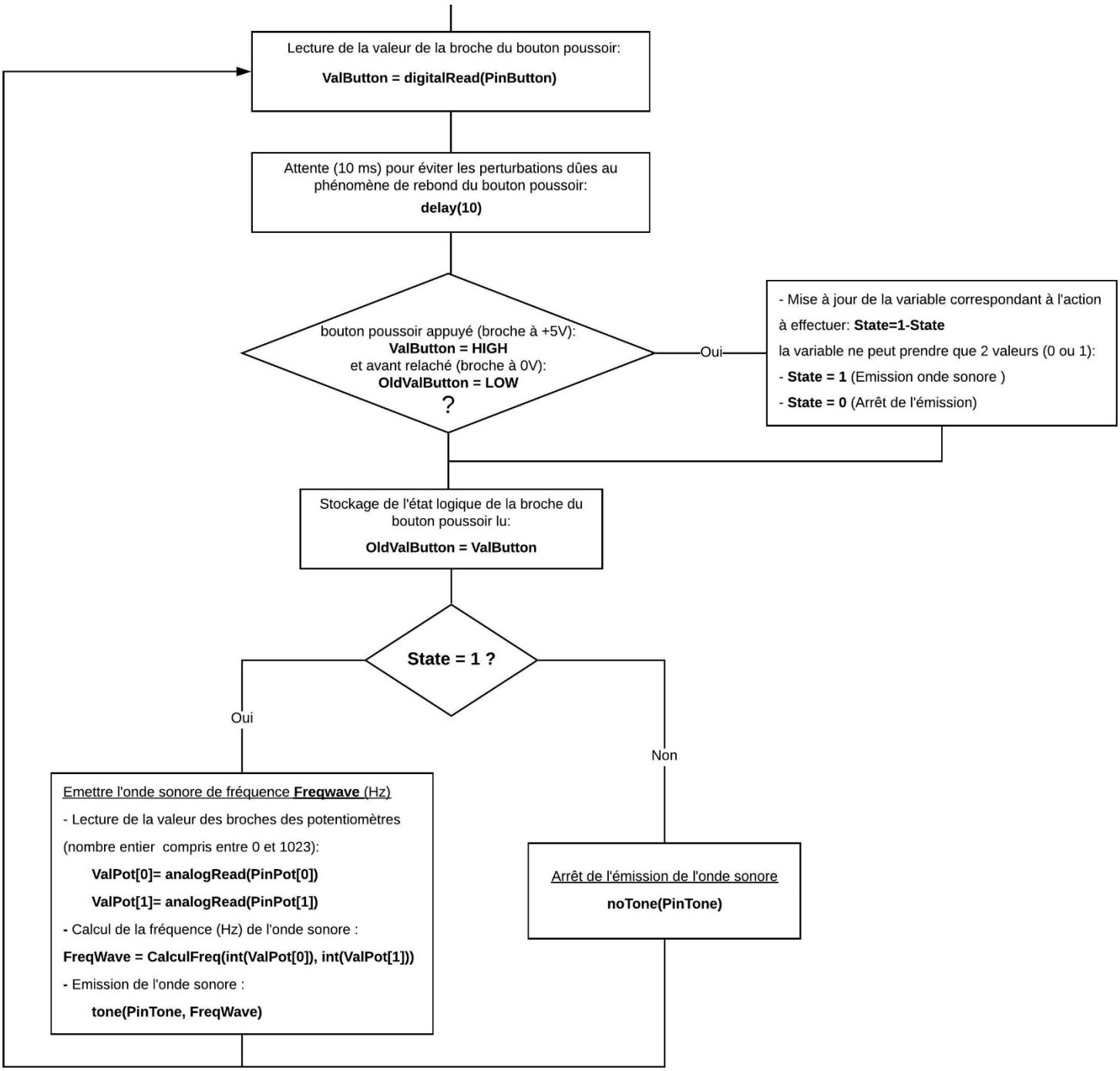
A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

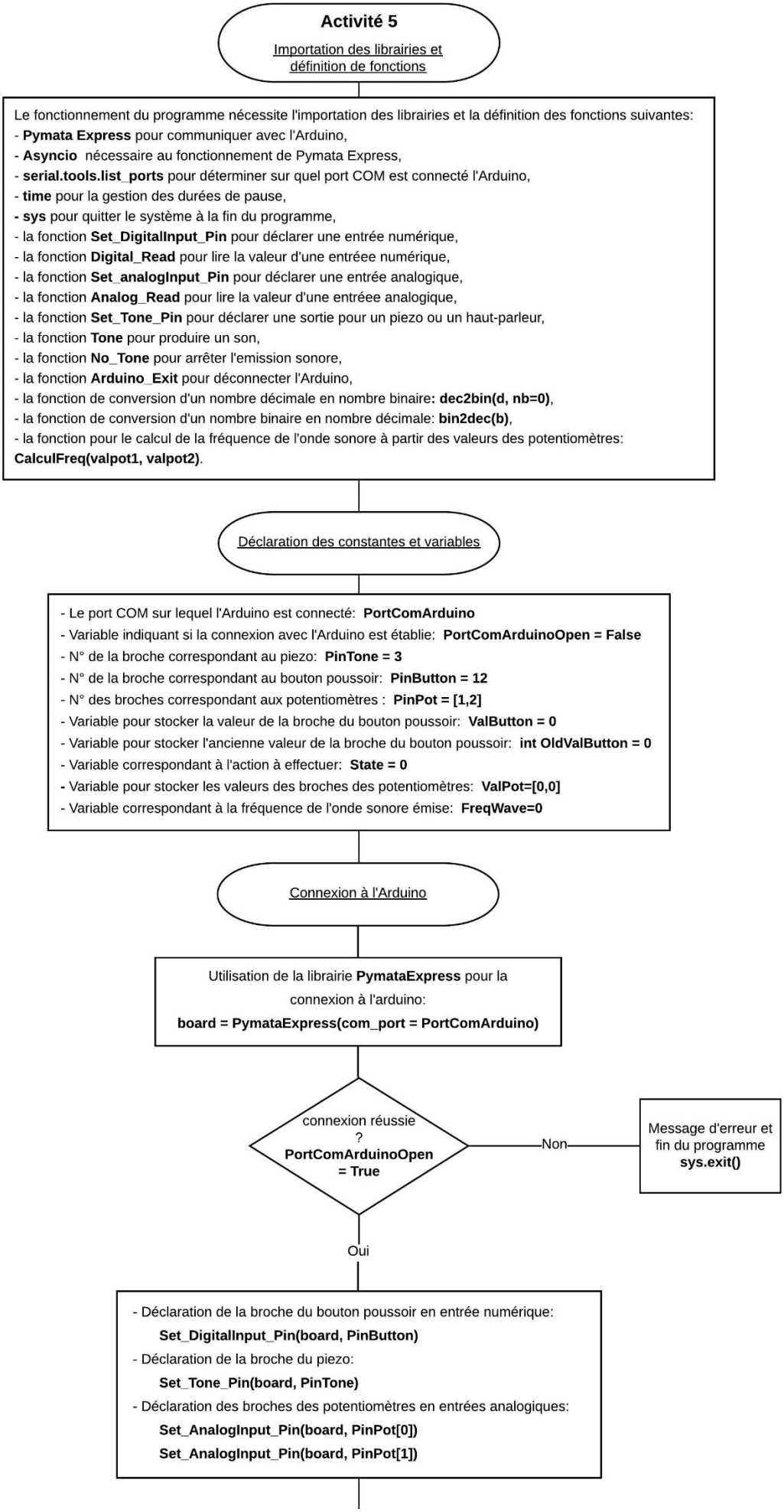
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

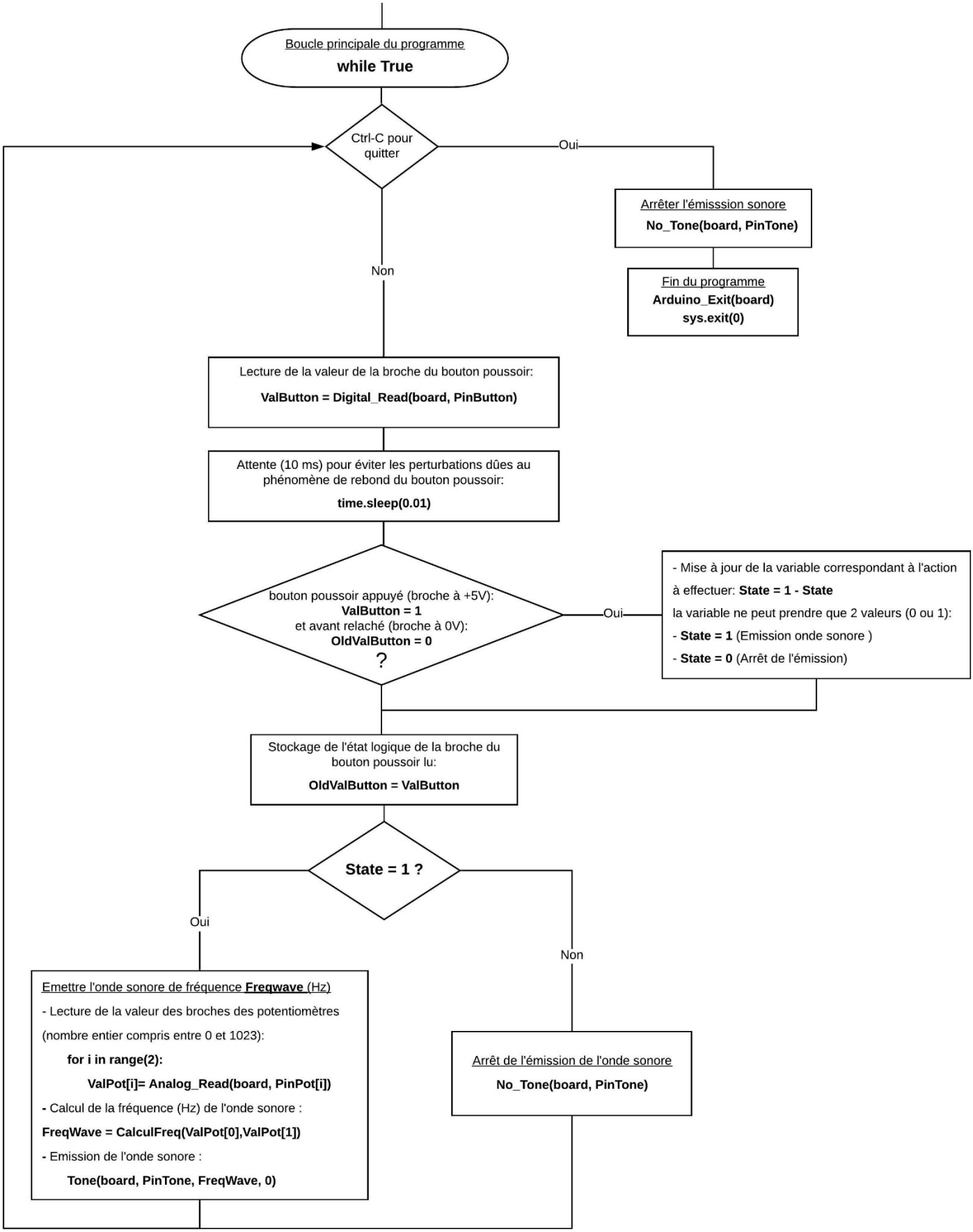
* Algorithme de programmation de l'activité 5 en langage Arduino IDE :




Lecture de la valeur de la broche du bouton poussoir:
ValButton = digitalRead(PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
delay(10)
bouton poussoir appuyé (broche à +5V):
ValButton = HIGH
et avant relaché (broche à 0V):
OldValButton = LOW
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1):
- State = 1 (Emission onde sonore)
- State = 0 (Arrêt de l'émission)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Emettre l'onde sonore de fréquence Freqwave (Hz)
- Lecture de la valeur des broches des potentiomètres (nombre entier compris entre 0 et 1023):
ValPot[0]= analogRead(PinPot[0])
ValPot[1]= analogRead(PinPot[1])
- Calcul de la fréquence (Hz) de l'onde sonore :
FreqWave = CalculFreq(int(ValPot[0]), int(ValPot[1]))
- Emission de l'onde sonore :
tone(PinTone, FreqWave)
Arrêt de l'émission de l'onde sonore
noTone(PinTone)

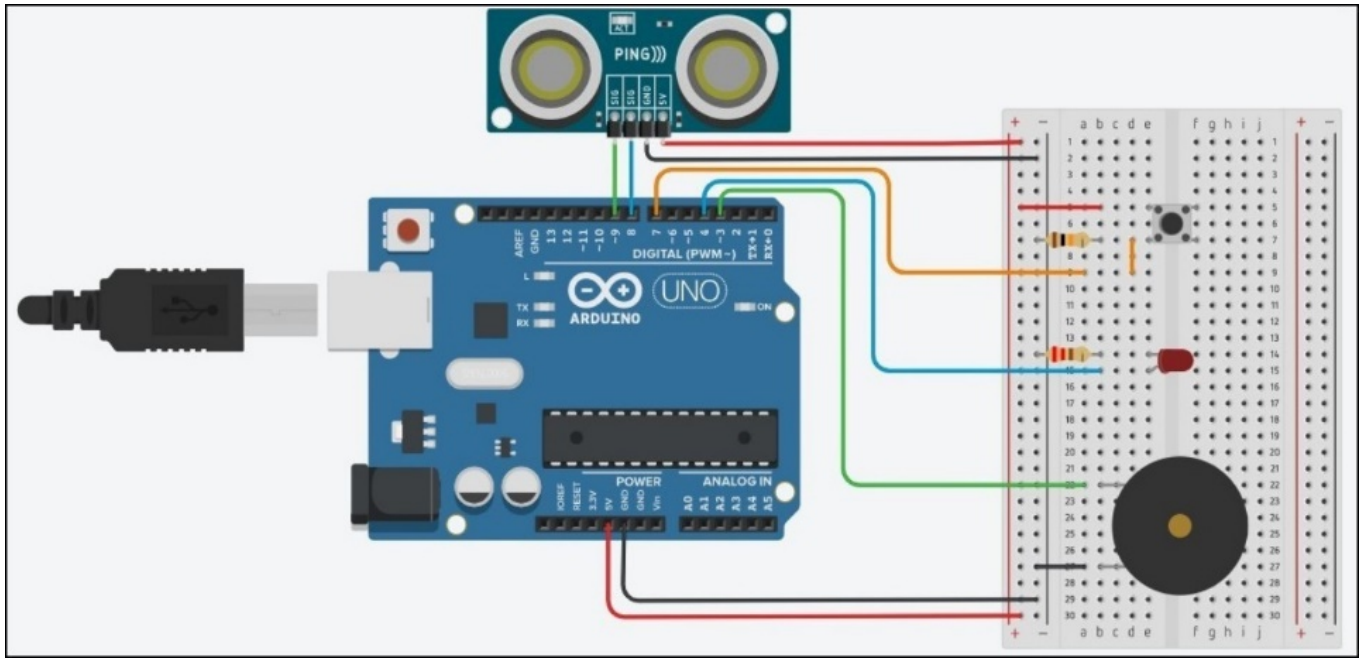
* Algorithme de programmation de l'activité 5 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Arrêter l'émisssion sonore
No_Tone(board, PinTone)
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Non
Lecture de la valeur de la broche du bouton poussoir:
ValButton = Digital_Read(board, PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(0.01)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State = 1 - State
la variable ne peut prendre que 2 valeurs (0 ou 1):
- State = 1 (Emission onde sonore)
- State = 0 (Arrêt de l'émission)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Oui
Non
Emettre l'onde sonore de fréquence Freqwave (Hz)
- Lecture de la valeur des broches des potentiomètres (nombre entier compris entre 0 et 1023):
for i in range(2):
ValPot[i]= Analog_Read(board, PinPot[i])
- Calcul de la fréquence (Hz) de l'onde sonore :
FreqWave = CalculFreq(ValPot[0],ValPot[1])
- Emission de l'onde sonore :
Tone(board, PinTone, FreqWave, 0)
Arrêt de l'émission de l'onde sonore
No_Tone(board, PinTone)

2.5.4 Ondes ultrasonores : Vitesse & Distances

Nous avons vu qu'il était possible, avec un Arduino, de produire des ondes sonores, caractérisées par leur fréquence. Nous allons maintenant, nous intéresser à la vitesse de propagation des ondes sonores.



- **Liste des composants :**

. 1 capteur ultrasonique (par exemple, le HC-SR04)
. 1 DEL Rouge
. 1 résistance de 220 Ω
. 1 résistance de 10 kΩ
. 1 bouton poussoir
. 1 haut-parleur (ou piezzo)
. 1 plaque d'essai
. Fils de connexion

- **Protocole de communication (Mode "Contrôle de l'Arduino") :**

. Firmata Express

Rappels :

Le son est une onde mécanique qui se propage dans un milieu matériel fluide (air, eau) ou solide et les ondes sonores sont caractérisées par leur fréquence.

Les sons audibles par l'Homme ont des fréquences comprises entre 20 et 20 000 Hz, les infrasons ont une fréquence inférieure à 20 Hz, et les ultrasons sont situés au-delà de 20 kHz.

La vitesse de propagation, ou célérité, du son est indépendante de sa fréquence mais dépend du milieu de propagation : plus le milieu matériel est dense plus la vitesse est grande.

Par exemple : c (air) = 340 $m.s^{-1}$; c (eau de mer) = 1 500 $m.s^{-1}$; c (acier) = 5 000 $m.s^{-1}$

La célérité du son dépend de la température, c'est-à-dire de l'agitation des particules qui constituent le milieu de propagation : plus la température est élevée plus le son se propage vite.

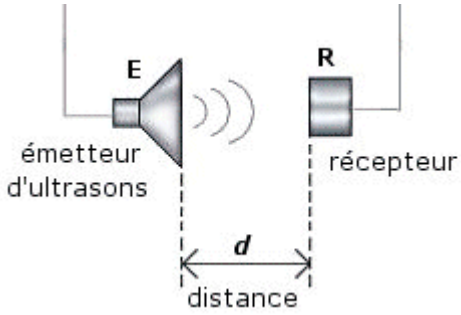
Par exemple : c (air à 0°C) = 331 $m.s^{-1}$; c (air à 15°C) = 340 $m.s^{-1}$

La relation entre la vitesse du son dans l'air en m/s et la température en kelvins est:

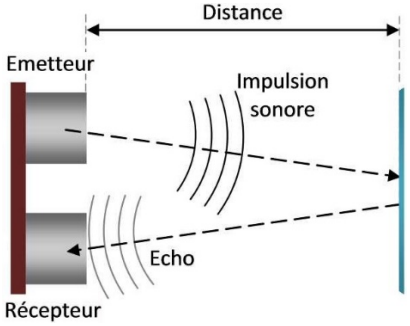
$$c_{\text{air}} = 20{,}05\sqrt{T}$$ (T en kelvins = T en °C + 273,15)

La célérité dans l'air, en $m.s^{-1}$, peut être déterminée expérimentalement en mesurant la durée de propagation Dt, en s, de l'onde sonore, entre un émetteur et un récepteur situés à une distance d, en m, grâce à la relation :

$$C_{air} = \frac{d}{Dt}$$



On peut également utiliser un ensemble émetteur - récepteur d'onde ultrasonores placé devant un obstacle. C'est le principe de la mesure par écho ou du Sonar :



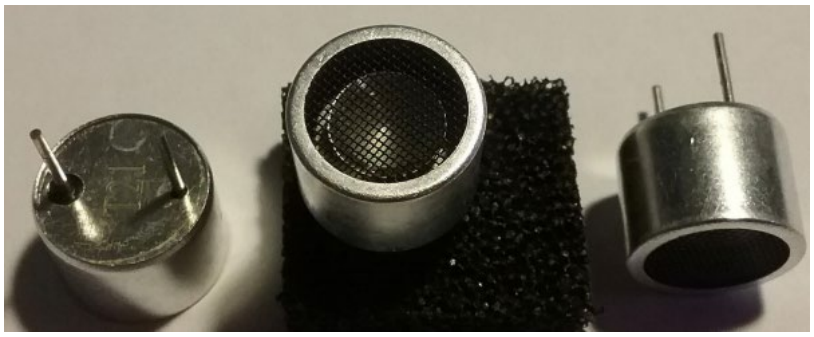
Dans ce cas, la distance parcourue par l'onde sonore, pendant la durée Dt, est 2 d, et alors :

$$C_{air} = \frac{2d}{Dt}$$

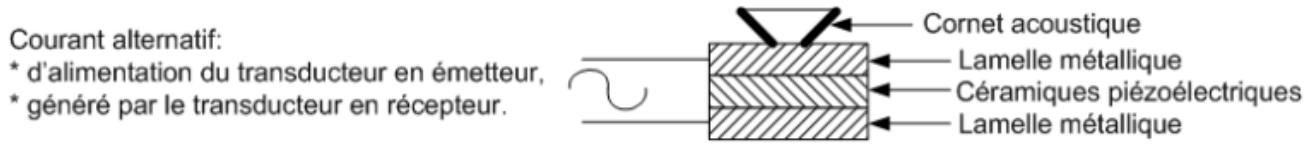
Principe de fonctionnement des émetteurs et récepteurs à ultrasons :

Les émetteurs et récepteurs à ultrasons sont aussi appelés transducteurs piézoélectriques, car ils convertissent une énergie électrique en énergie mécanique et réciproquement. Le principe de fonctionnement est donc identique à celui des buzzer.

Extérieurement les transducteurs d'émission (généralement, repérés par un « T » gravé) sont très semblables à ceux de réception (généralement, repérés par un « R » gravé) :



Un schéma de principe de ces transducteurs est donné ci-dessous :



. Fonctionnement d'un émetteur US :

Le transducteur est alimenté par une tension alternative à une fréquence nominale de fonctionnement (souvent 40 KHz). Cette tension, est appliquée sur les lamelles métalliques ce qui génère une déformation mécanique des céramiques qui est transformée en pression acoustique appliquée à l'air ambiant, via le cornet acoustique.

. Fonctionnement d'un récepteur US :

La pression acoustique (due à l'onde ultrasonore) reçue à travers l'air ambiant, via le cornet acoustique du récepteur US, est transformée en contrainte mécanique dans les céramiques qui génèrent des charges électriques sur les lamelles métalliques et donc une tension alternative à ses bornes.

Avec un Arduino, l'ensemble émetteur - récepteur d'onde ultrasonores ou le capteur ultrasonique le plus couramment utilisé est le HC-SR04.

Capteur ultrasonique HC-SR04

Caractéristiques

Le capteur est composé d'un émetteur d'ultrasons, d'un récepteur et du circuit de commande. Il est généralement utilisé pour mesurer des distances entre le capteur et un obstacle.

- Dimensions : 45 mm x 20 mm x 15 mm
- Plage de mesure : 2 cm à 400 cm
- Résolution de la mesure annoncée : 0,3 cm (en pratique : 1 cm)
- Angle de mesure efficace : 15 °

Broches de connexion

- Vcc = Alimentation +5 V DC
- Trig = Entrée émetteur d'impulsion d'ultrasons (Trigger input)
- Echo = Sortie récepteur d'impulsion d'ultrasons (Echo output)
- GND = Masse 0V

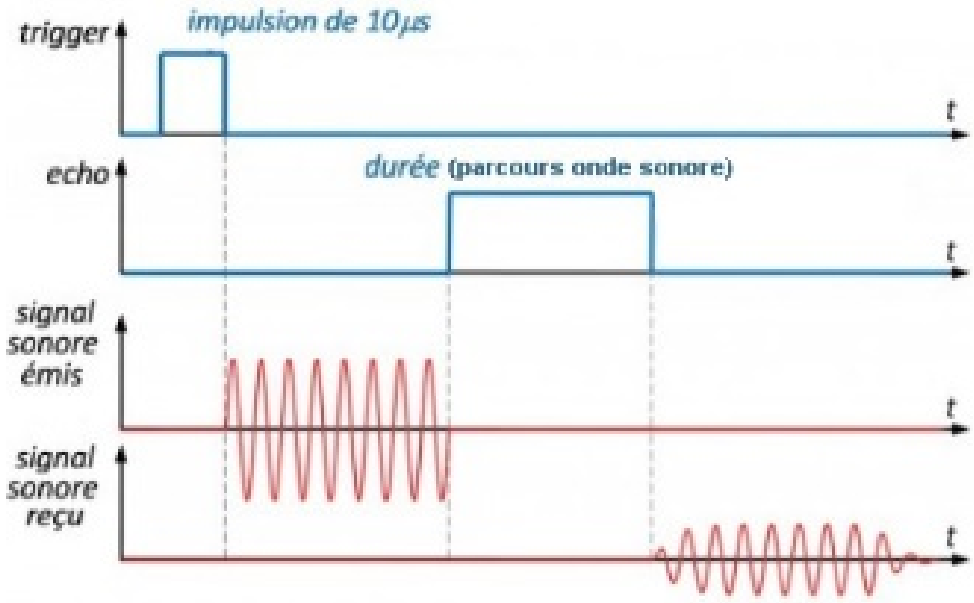
Spécifications et limites

- Tension d'alimentation : 5.0 V à ± 0.5 V
- Courant de repos : 2.0 mA à ± 0.5 mA
- Courant de fonctionnement : 15 ± 5 mA
- Fréquence des ultrasons : 40 kHz

Le principe de fonctionnement :

1. Envoyer un signal numérique à l'état haut sur l'émetteur pendant 10 µs,
2. Le capteur envoie automatiquement 8 impulsions d'ultrasons à 40 kHz,
3. A la fin des 8 impulsions, la sortie Echo du capteur passe à l'état haut,
4. Si le signal revient et est détecté par le récepteur, la sortie Echo du capteur passe à l'état bas. La durée de l'état haut du signal Echo correspond au temps entre l'émission des ultrasons et leur réception.

Le principe de fonctionnement est résumé sur le schéma suivant :



La formule couramment utilisée dans les programmes Arduino permettant de calculée la distance entre le capteur et un obstacle est :

Distance $_{\text{capteur- obstacle}}$ **(en cm) = durée propagation (en µs) / 58**

En effet, pour cela, on suppose que la vitesse des ultrasons dans l'air est de V = 340 m.s^{-1}, la distance parcourue, **d** (en m), par l'onde sonore pendant la durée, **Dt** (en s), est alors :

$$\mathbf{d_{parcours\ onde\ sonore} = V_{ultrasons}\ x\ Dt}$$

Soit : Distance $_{\text{capteur- obstacle}}$ (en m) = d $_{\text{parcours onde sonore}}$ /2 = $V_{ultrasons}$ x Dt / 2

Distance $_{\text{capteur- obstacle}}$ (en m) = 340 x Dt / 2

Distance $_{\text{capteur- obstacle}}$ (en cm) = 34000 x Dt (en µs) /2000000 = 17x Dt (en µs) / 1000

Distance $_{\text{capteur- obstacle}}$ **(en cm) = Dt (en µs) / 58,82**

Le capteur ultrasonique **HC-SR04** dispose de 2 broches différentes pour l'émission et la réception des ultrasons.

Il existe également des capteurs ultrasoniques à une seule broche, comme le **Grove 101020010**

Capteur ultrasonique Grove 101020010

Caractéristiques

Le capteur est composé d'un émetteur d'ultrasons, d'un récepteur et du circuit de commande. Il est généralement utilisé pour mesurer des distances entre le capteur et un obstacle.

- Dimensions : 50 mm x 25 mm x 16 mm
- Plage de mesure : 2 cm à 350 cm
- Résolution de la mesure : 1 cm
- Angle de mesure efficace : 15 °

Broches de connexion

- Vcc = Alimentation +5 V DC
- SIG = Entrée émetteur et sortie récepteur d'impulsion d'ultrasons
- GND = Masse 0V
- NC = Non connectée

Spécifications et limites

- Tension d'alimentation : 3,2 V à 5,2 V
- Courant de fonctionnement : 8 mA
- Fréquence des ultrasons : 40 kHz
- Température de fonctionnement : 10 - 60 °C

Le principe de fonctionnement :

1. Déclarer la broche de l'Arduino sur laquelle est connectée la broche **SIG** du capteur en sortie numérique et la maintenir à l'état bas pendant 2 µs,

2. Envoyer un signal numérique à l'état haut sur l'émetteur (sur la broche SIG) pendant 5 µs,

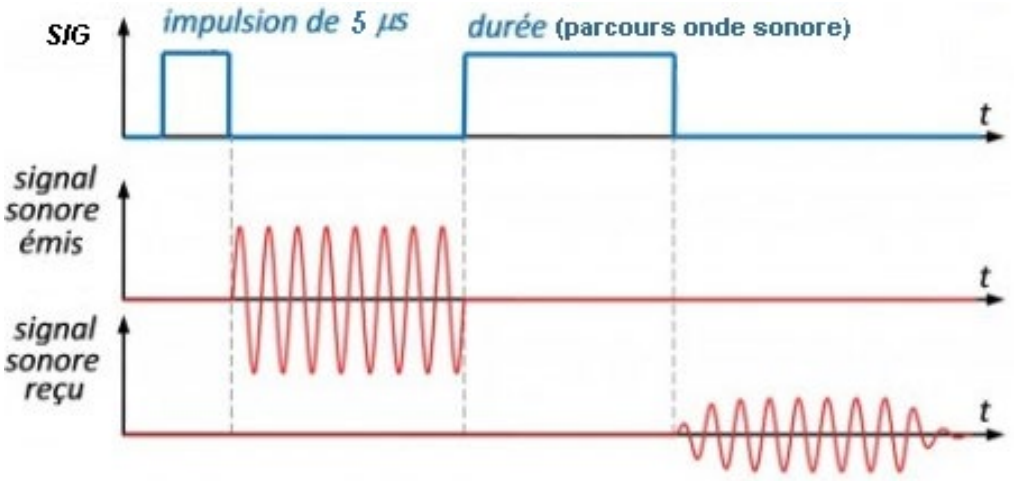
3. Le capteur envoie automatiquement des impulsions d'ultrasons à 40 kHz,

4. Déclarer la broche de l'Arduino sur laquelle est connectée la broche **SIG** du capteur en entrée numérique, afin de pouvoir recevoir le signal du récepteur d'ultrasons.

5. A la fin des impulsions, la broche **SIG** du capteur passe à l'état haut,

6. Si le signal revient et est détecté par le récepteur, la broche **SIG** du capteur passe à l'état bas. La durée de l'état haut du signal **SIG** correspond au temps entre l'émission des ultrasons et leur réception.

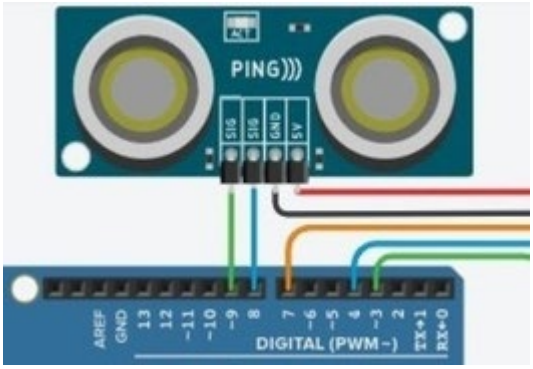
Le principe de fonctionnement est résumé sur le schéma suivant :



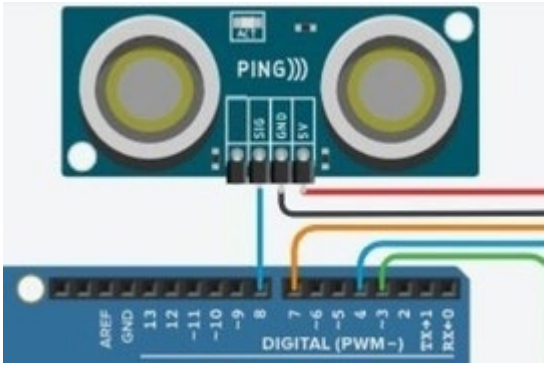
Le principe de calcul de la distance entre ce capteur et un obstacle reste le même.

Par défaut, ARDUINO LAB est configuré pour fonctionner avec un capteur ultrasonique dont les broches pour l'émission et la réception des ultrasons sont distinctes.

Pour utiliser un capteur ultrasonique qui ne dispose que d'une broche pour le signal d'émission ou de réception, il suffit de cliquer sur la broche 9 avant de connecter l'Arduino. Un nouveau clic sur la broche 9 permet de revenir à un capteur à 2 broches :



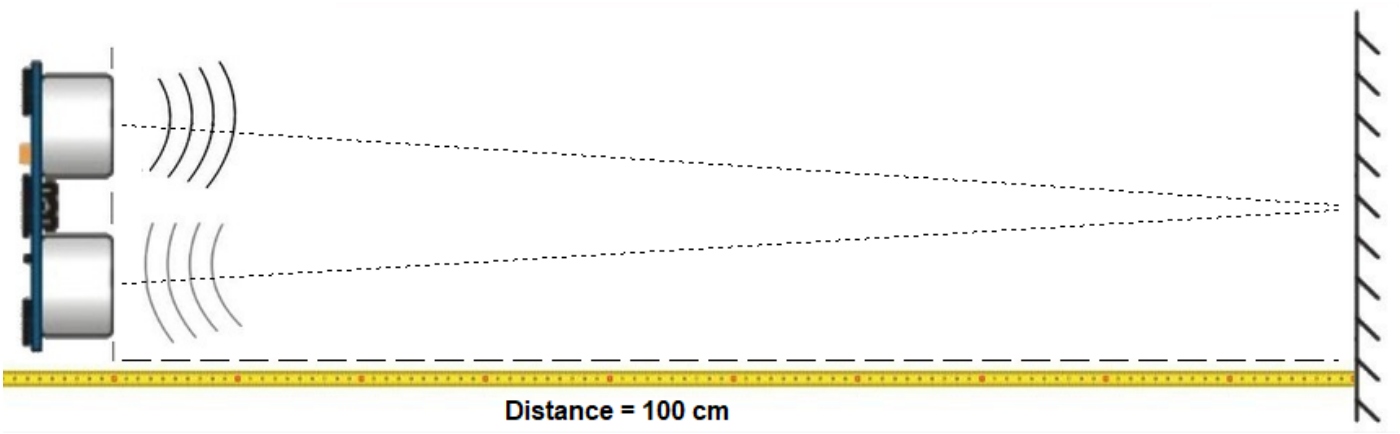
Capteur à 2 broches différentes Trigg et Echo



Capteur à 1 broche commune pour Trigg et Echo

- Activité 1 : Détermination de la vitesse du son dans l'air

Dans cette activité, nous allons déterminer expérimentalement la vitesse de propagation des ondes sonores en mesurant, à l'aide d'un capteur à ultrasons (par exemple, le HC-SR04), la durée de propagation, **Dt**, de l'onde sonore entre l'émetteur et le récepteur situés à une distance, **d**, connue d'un obstacle.

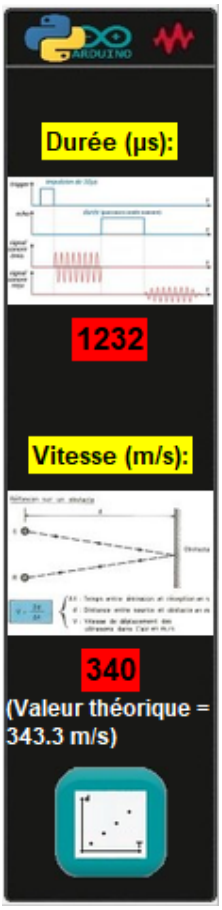


Après avoir cliqué sur la prise USB, un menu permettant de régler la distance entre le capteur et l'obstacle, et la température de l'air, est affiché.

Si le mode de fonctionnement est le "contrôle de l'Arduino", un appui sur le bouton poussoir réel ou virtuel déclenche la mesure de la durée de propagation de l'onde ultrasonore. Celle-ci est affichée, dans le menu "Résultats".
La vitesse de propagation en m/s est alors calculée et affichée. La vitesse théorique du son dans l'air en fonction de la température indiquée est également affichée afin de pouvoir la comparer à la valeur déterminée expérimentalement.

En mode "simulation", ARDUINO LAB utilise la valeur théorique de la vitesse de propagation des sons dans l'air à la température indiquée par l'utilisateur et affiche la durée de propagation de l'onde calculée avec cette valeur.

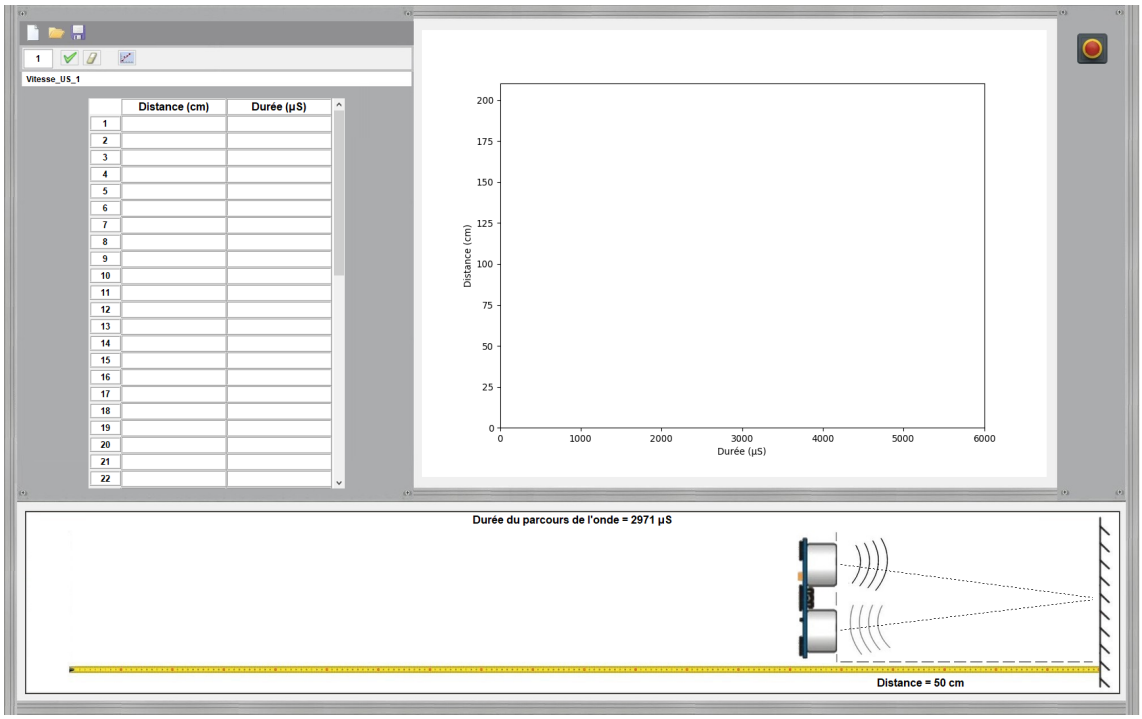
Menu "Réglages"



Menu "Résultats"

Il est possible d'enregistrer plusieurs durées de propagation pour différentes distances et de représenter graphiquement la distance en fonction de la durée afin de déterminer une vitesse moyenne du son dans l'air en cliquant sur :

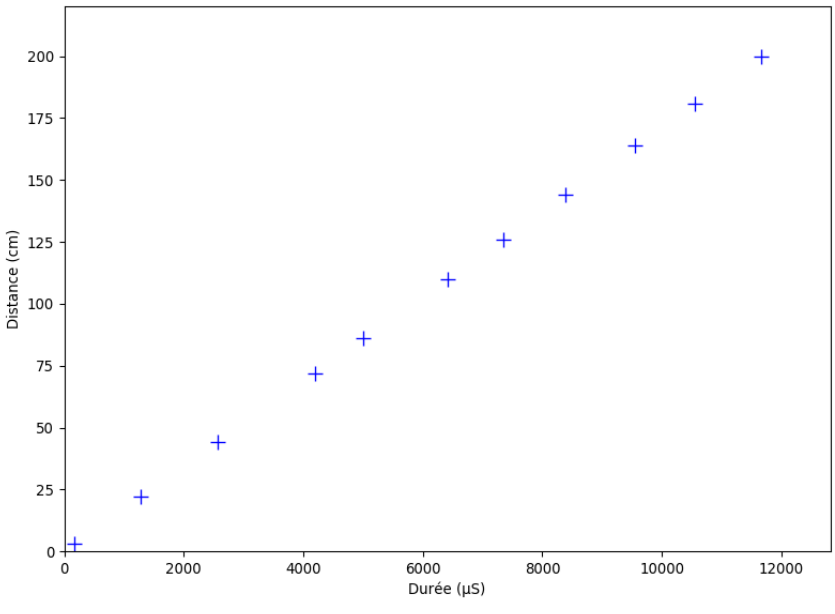
Une nouvelle fenêtre est alors affichée :



En mode "contrôle de l'Arduino", il suffit de positionner le capteur ultrasonique virtuel à la même distance que le capteur réel, la durée de propagation de l'onde est affichée, puis afin d'enregistrer les valeurs, de cliquer sur :  (pour effacer un point, cliquer sur : )

Les valeurs de distance et de durée de propagation sont affichées dans le tableur et le graphe est tracé automatiquement.

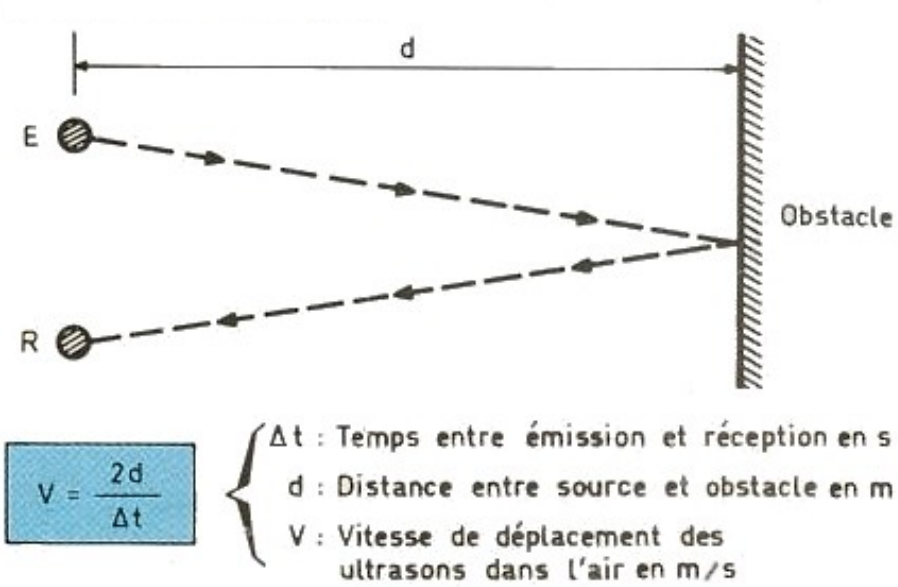
	Distance (cm)	Durée (µS)
1	3	174
2	22	1281
3	44	2563
4	72	4194
5	86	5010
6	110	6408
7	126	7340
8	144	8389
9	164	9554
10	181	10545
11	200	11652
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		



En mode "simulation", la durée de propagation est calculée, en utilisant la valeur théorique de la vitesse des sons dans l'air à la température indiquée et la distance choisie par l'utilisateur.

La modélisation de la distance en fonction de la durée est réalisée en appuyant sur : 

On rappelle que la vitesse de propagation de l'onde ultrasonore est :



La modélisation de **d = f(Dt)** est alors : **y = a x** (régression linéaire)

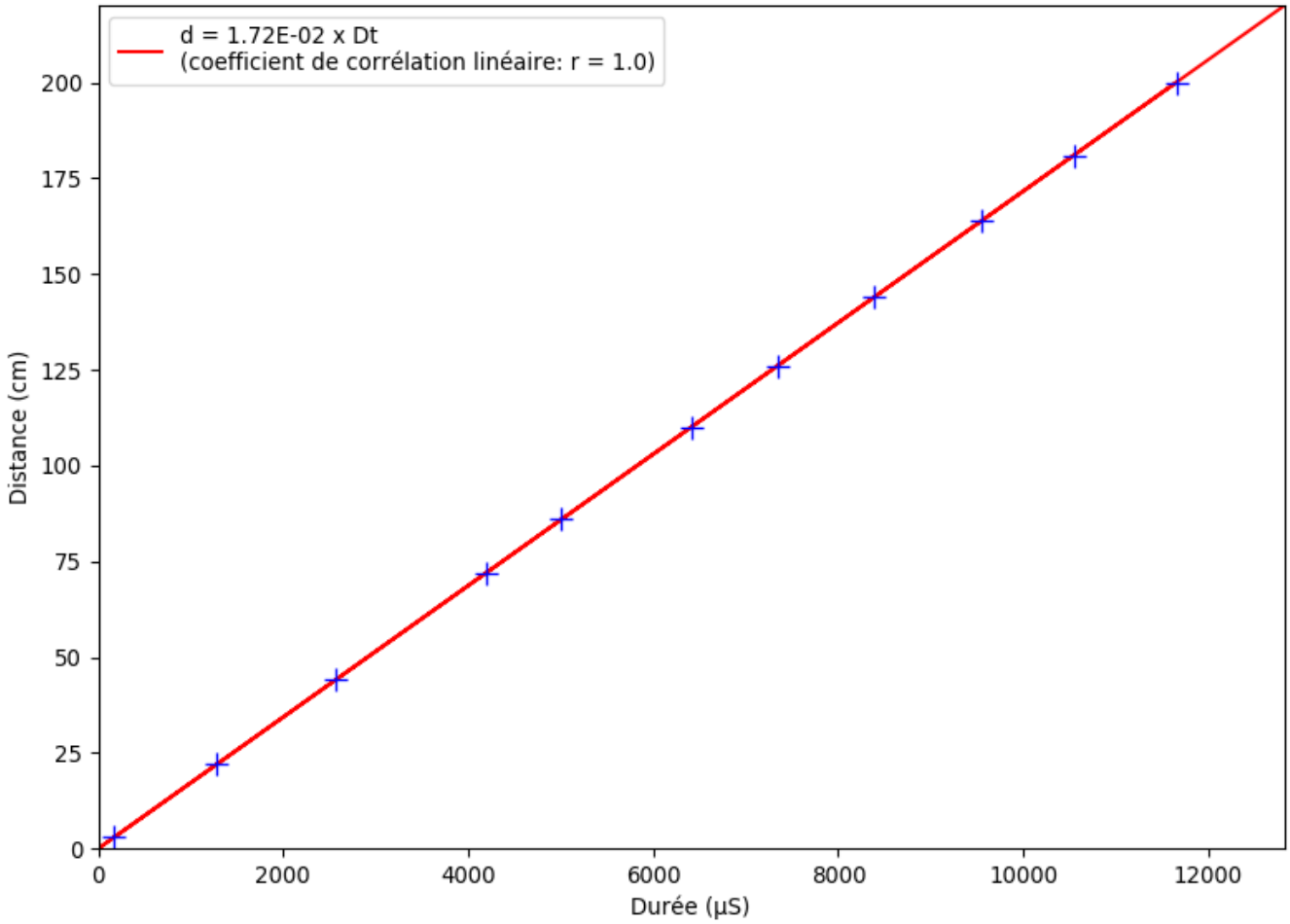
Où **a** est le coefficient directeur de la droite, avec : **a = V / 2**

On en déduit alors : **V = 2 a**

<u>Attention</u> :

Les mesures effectuées sont exprimées en cm pour la distance et en µS pour la durée. La vitesse calculée est donc en **cm/µS**. Il faut multiplier le résultat par 10000 pour obtenir une vitesse en **m/s**.

La droite de modélisation est alors affichée :



. Les données peuvent être enregistrées dans un fichier csv en cliquant sur : 

. Un fichier de mesures est ouvert en cliquant sur : 

. Un nouveau fichier de mesure est créé en cliquant sur : 

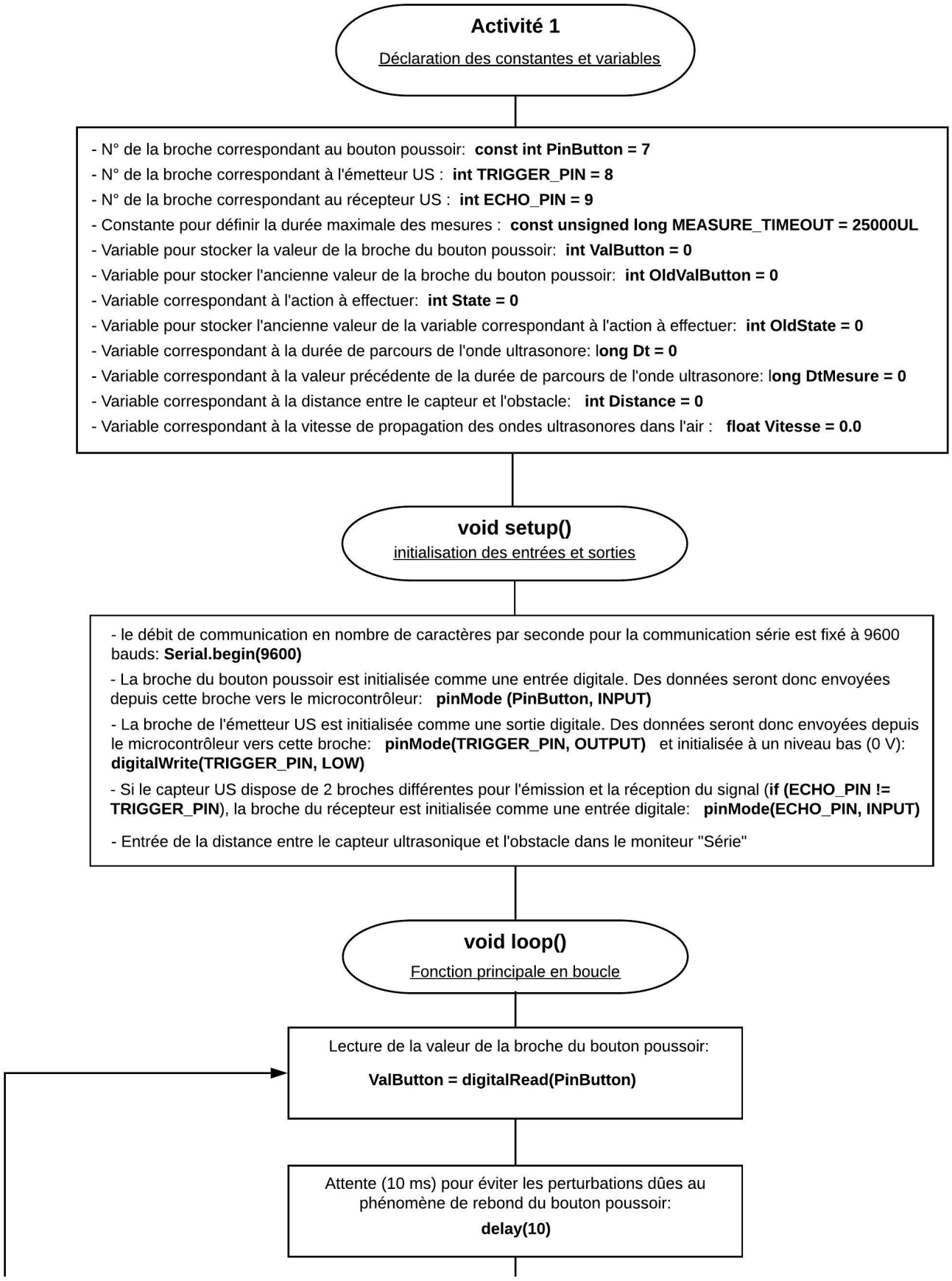
La fenêtre "Détermination de la vitesse du son dans l'air" est fermée en cliquant sur : 

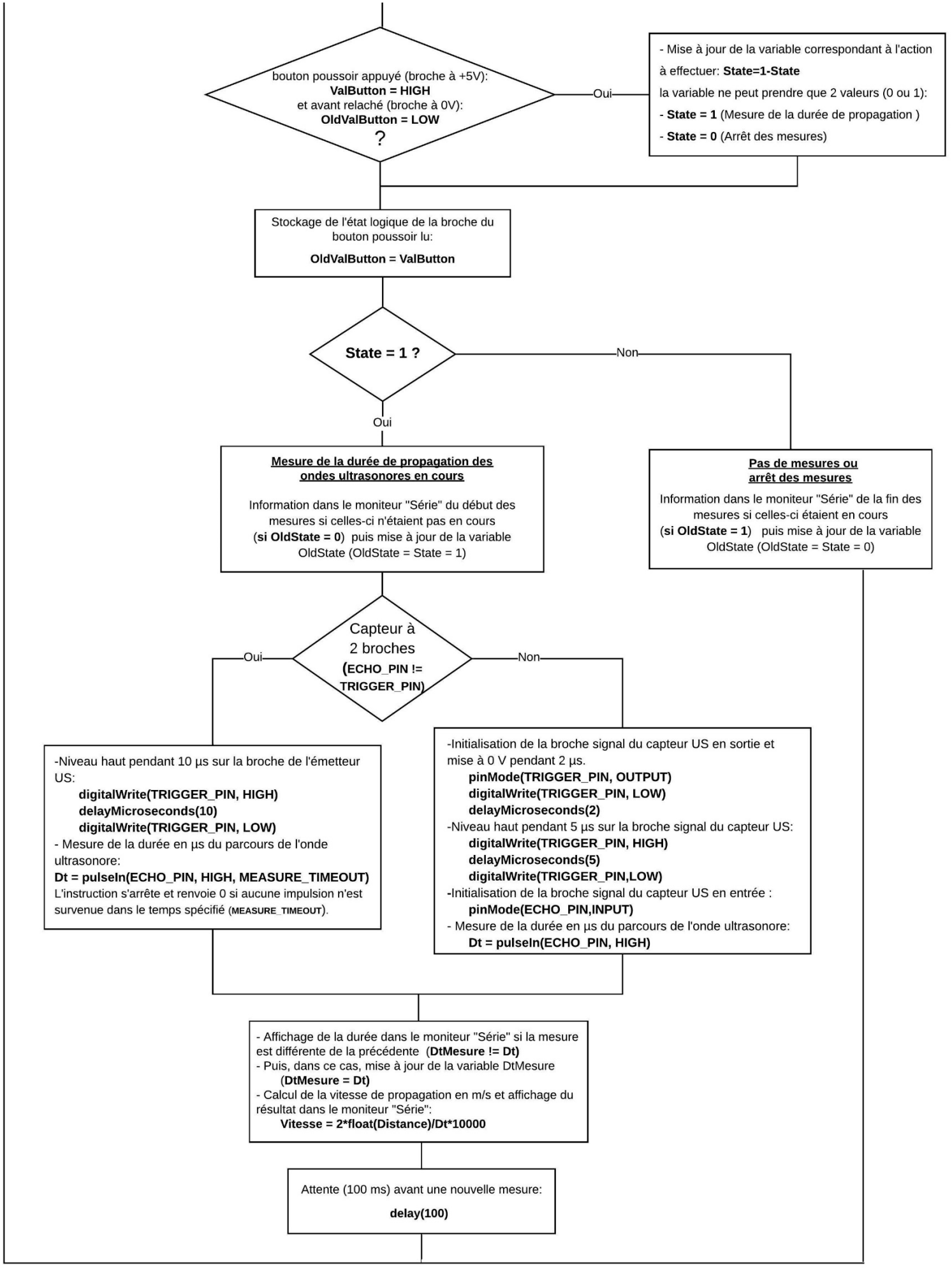
Les mesures sont arrêtées en appuyant de nouveau sur le bouton poussoir réel ou virtuel, en mode "Contrôle de l'Arduino", ou virtuel en mode "Simulation".

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

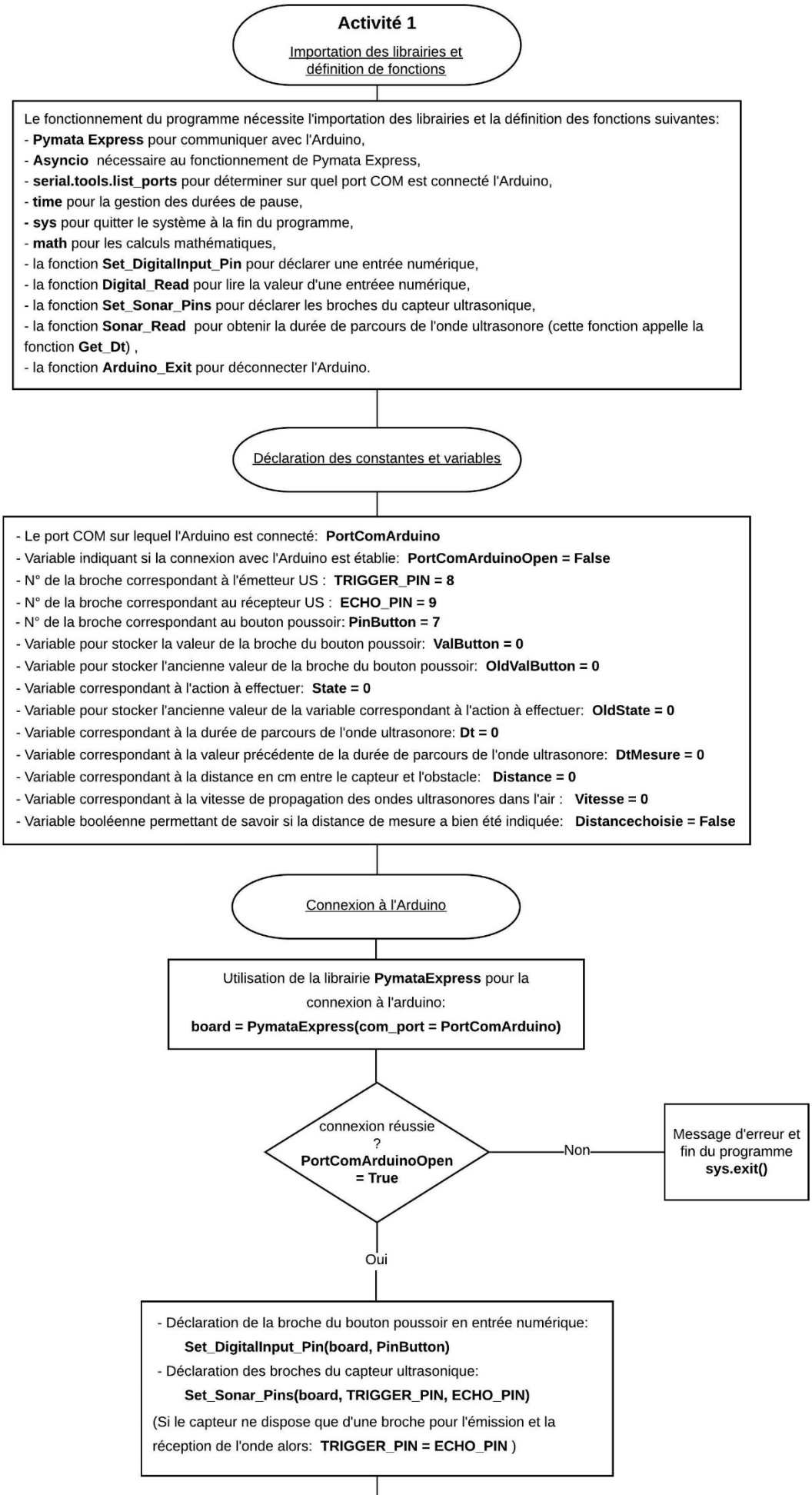
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

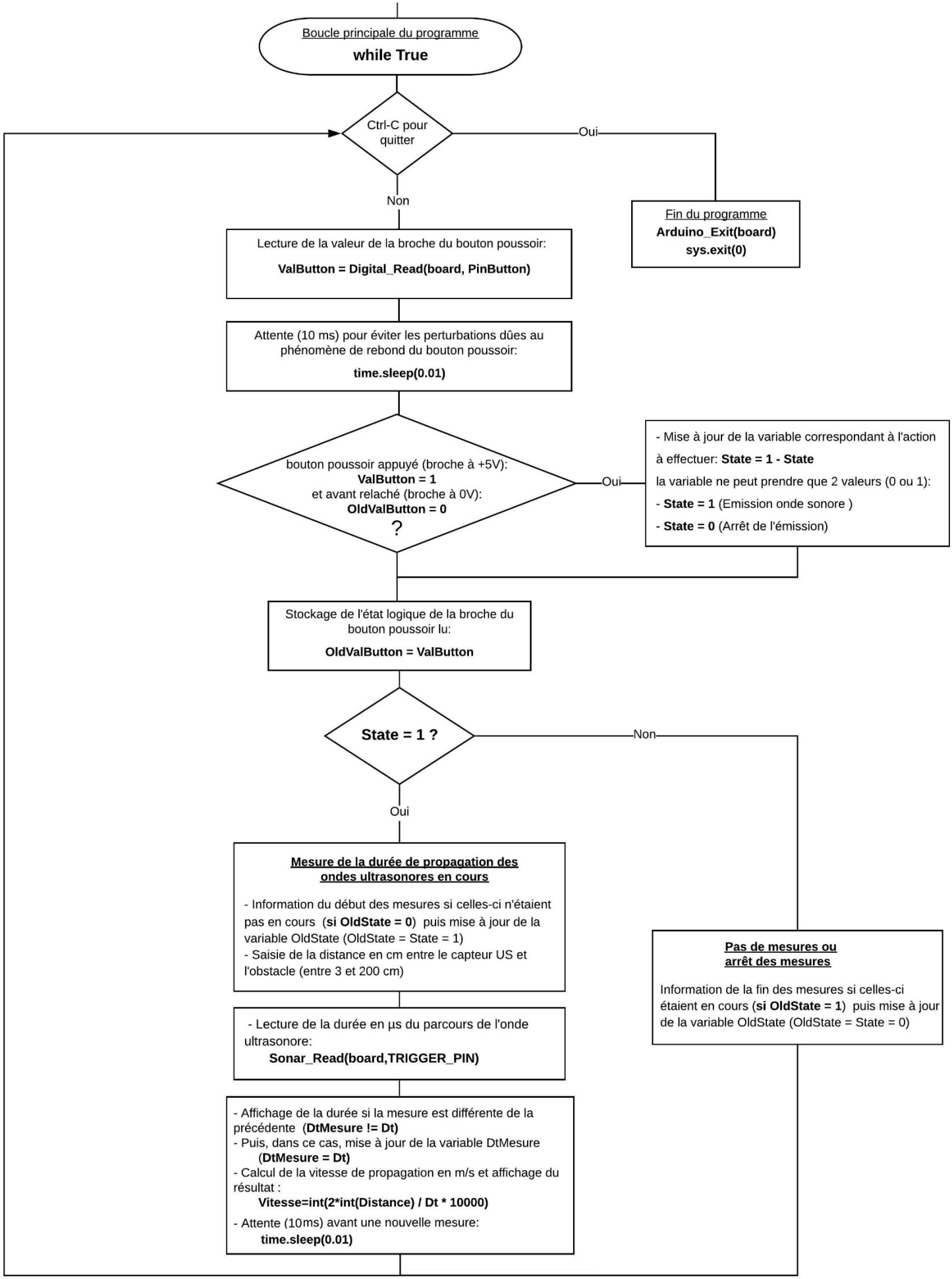
* Algorithme de programmation de l'activité 1 en langage Arduino IDE :




bouton poussoir appuyé (broche à +5V):
ValButton = HIGH
et avant relaché (broche à 0V):
OldValButton = LOW
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State=1-State
la variable ne peut prendre que 2 valeurs (0 ou 1):
- State = 1 (Mesure de la durée de propagation)
- State = 0 (Arrêt des mesures)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Non
Oui
Mesure de la durée de propagation des ondes ultrasonores en cours
Information dans le moniteur "Série" du début des mesures si celles-ci n'étaient pas en cours (si OldState = 0) puis mise à jour de la variable OldState (OldState = State = 1)
Pas de mesures ou arrêt des mesures
Information dans le moniteur "Série" de la fin des mesures si celles-ci étaient en cours (si OldState = 1) puis mise à jour de la variable OldState (OldState = State = 0)
Capteur à 2 broches (ECHO_PIN != TRIGGER_PIN)
Oui
Non
-Niveau haut pendant 10 µs sur la broche de l'émetteur US:
digitalWrite(TRIGGER_PIN, HIGH)
delayMicroseconds(10)
digitalWrite(TRIGGER_PIN, LOW)
- Mesure de la durée en µs du parcours de l'onde ultrasonore:
Dt = pulseIn(ECHO_PIN, HIGH, MEASURE_TIMEOUT)
L'instruction s'arrête et renvoie 0 si aucune impulsion n'est survenue dans le temps spécifié (MEASURE_TIMEOUT).
-Initialisation de la broche signal du capteur US en sortie et mise à 0 V pendant 2 µs.
pinMode(TRIGGER_PIN, OUTPUT)
digitalWrite(TRIGGER_PIN, LOW)
delayMicroseconds(2)
-Niveau haut pendant 5 µs sur la broche signal du capteur US:
digitalWrite(TRIGGER_PIN, HIGH)
delayMicroseconds(5)
digitalWrite(TRIGGER_PIN,LOW)
-Initialisation de la broche signal du capteur US en entrée :
pinMode(ECHO_PIN,INPUT)
- Mesure de la durée en µs du parcours de l'onde ultrasonore:
Dt = pulseIn(ECHO_PIN, HIGH)
- Affichage de la durée dans le moniteur "Série" si la mesure est différente de la précédente (DtMesure != Dt)
- Puis, dans ce cas, mise à jour de la variable DtMesure (DtMesure = Dt)
- Calcul de la vitesse de propagation en m/s et affichage du résultat dans le moniteur "Série":
Vitesse = 2*float(Distance)/Dt*10000
Attente (100 ms) avant une nouvelle mesure:
delay(100)

* Algorithme de programmation de l'activité 1 en Python :




Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Lecture de la valeur de la broche du bouton poussoir:
ValButton = Digital_Read(board, PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(0.01)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State = 1 - State
la variable ne peut prendre que 2 valeurs (0 ou 1):
- State = 1 (Emission onde sonore)
- State = 0 (Arrêt de l'émission)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Non
Oui
Mesure de la durée de propagation des ondes ultrasonores en cours
- Information du début des mesures si celles-ci n'étaient pas en cours (si OldState = 0) puis mise à jour de la variable OldState (OldState = State = 1)
- Saisie de la distance en cm entre le capteur US et l'obstacle (entre 3 et 200 cm)
Pas de mesures ou arrêt des mesures
Information de la fin des mesures si celles-ci étaient en cours (si OldState = 1) puis mise à jour de la variable OldState (OldState = State = 0)
- Lecture de la durée en µs du parcours de l'onde ultrasonore:
Sonar_Read(board,TRIGGER_PIN)
- Affichage de la durée si la mesure est différente de la précédente (DtMesure != Dt)
- Puis, dans ce cas, mise à jour de la variable DtMesure (DtMesure = Dt)
- Calcul de la vitesse de propagation en m/s et affichage du résultat :
Vitesse=int(2*int(Distance) / Dt * 10000)
- Attente (10ms) avant une nouvelle mesure:
time.sleep(0.01)

- Activité 2 : Mesure de distances

Maintenant que nous avons déterminer la célérité des ondes sonores dans l'air, dans cette activité, nous allons utiliser le capteur à ultrasons pour mesurer des distances, selon le même principe que l'activité précédente, en mesurant la durée de propagation, Dt, de l'onde sonore entre l'émetteur et le récepteur situés à une distance, d, inconnue d'un obstacle. C'est le principe du Sonar.

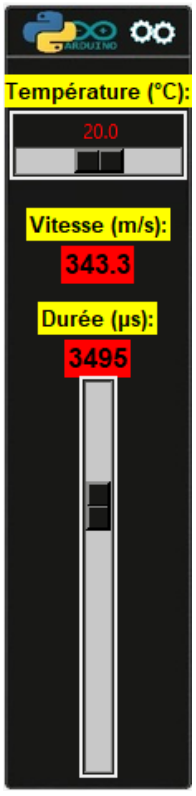
Si le mode de fonctionnement est le "contrôle de l'Arduino", après avoir cliqué sur la prise USB, un menu permettant de régler la température de l'air est affiché. La vitesse théorique du son dans l'air en fonction de la température indiquée est alors calculée et affichée.

Un appui sur le bouton poussoir réel ou virtuel déclenche la mesure de la durée de propagation, Dt, de l'onde ultrasonore.
La distance, en cm, entre le capteur et l'obstacle est alors calculée, à partir de Dt et de la célérité théorique du son dans l'air, et affichée dans le menu "Résultats".

Menu "Réglages"

Menu "Résultats"

En mode "simulation", l'utilisateur doit régler la durée de propagation, Dt, des ondes ultrasonores dans le menu "Réglages" et après avoir appuyé sur le bouton poussoir virtuel, la distance, en cm, correspondant à ce Dt est calculée et affichée le menu "Résultats".

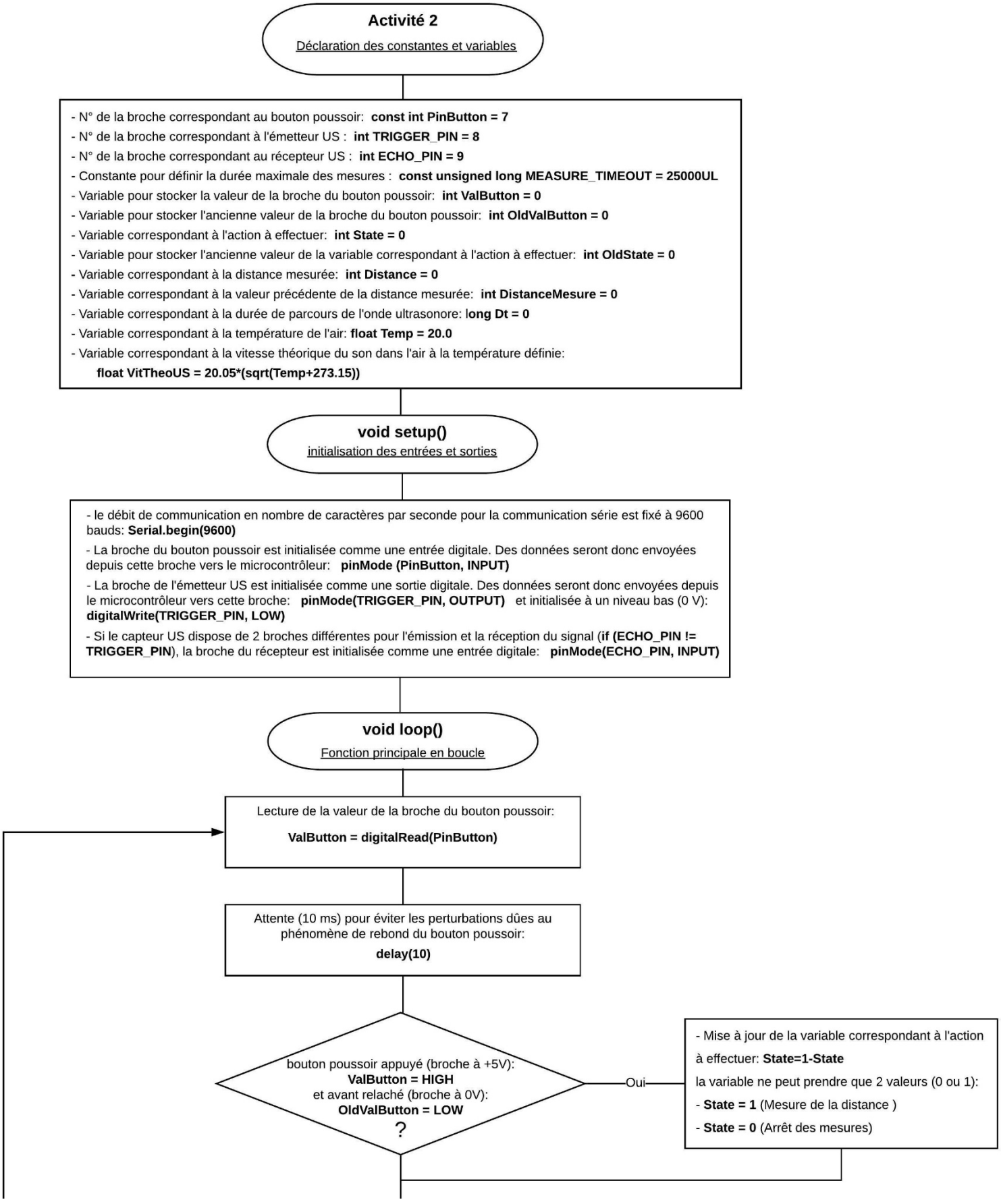


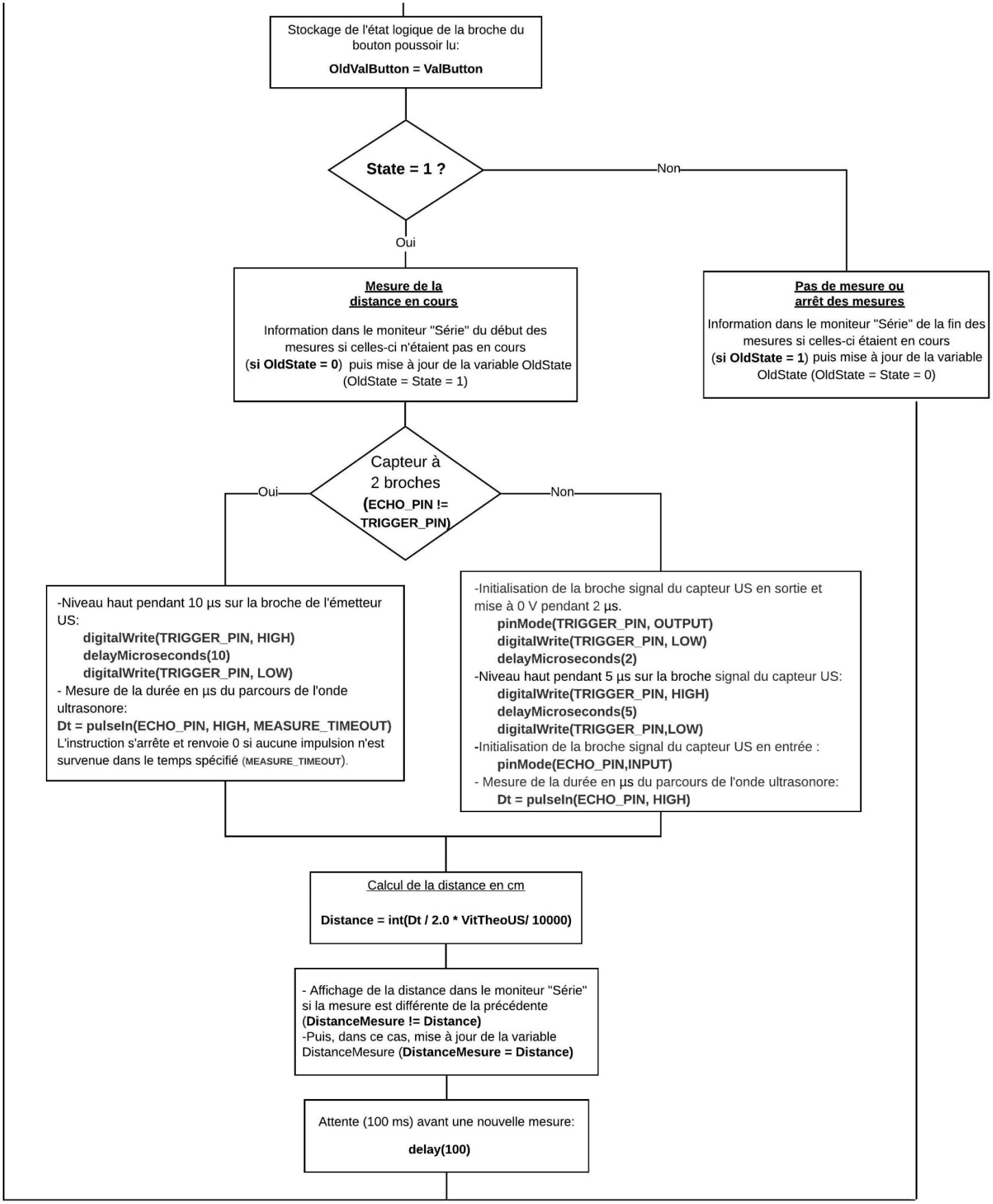
Les mesures sont arrêtées en appuyant de nouveau sur le bouton poussoir réel ou virtuel, en mode "Contrôle de l'Arduino", ou virtuel en mode "Simulation".

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

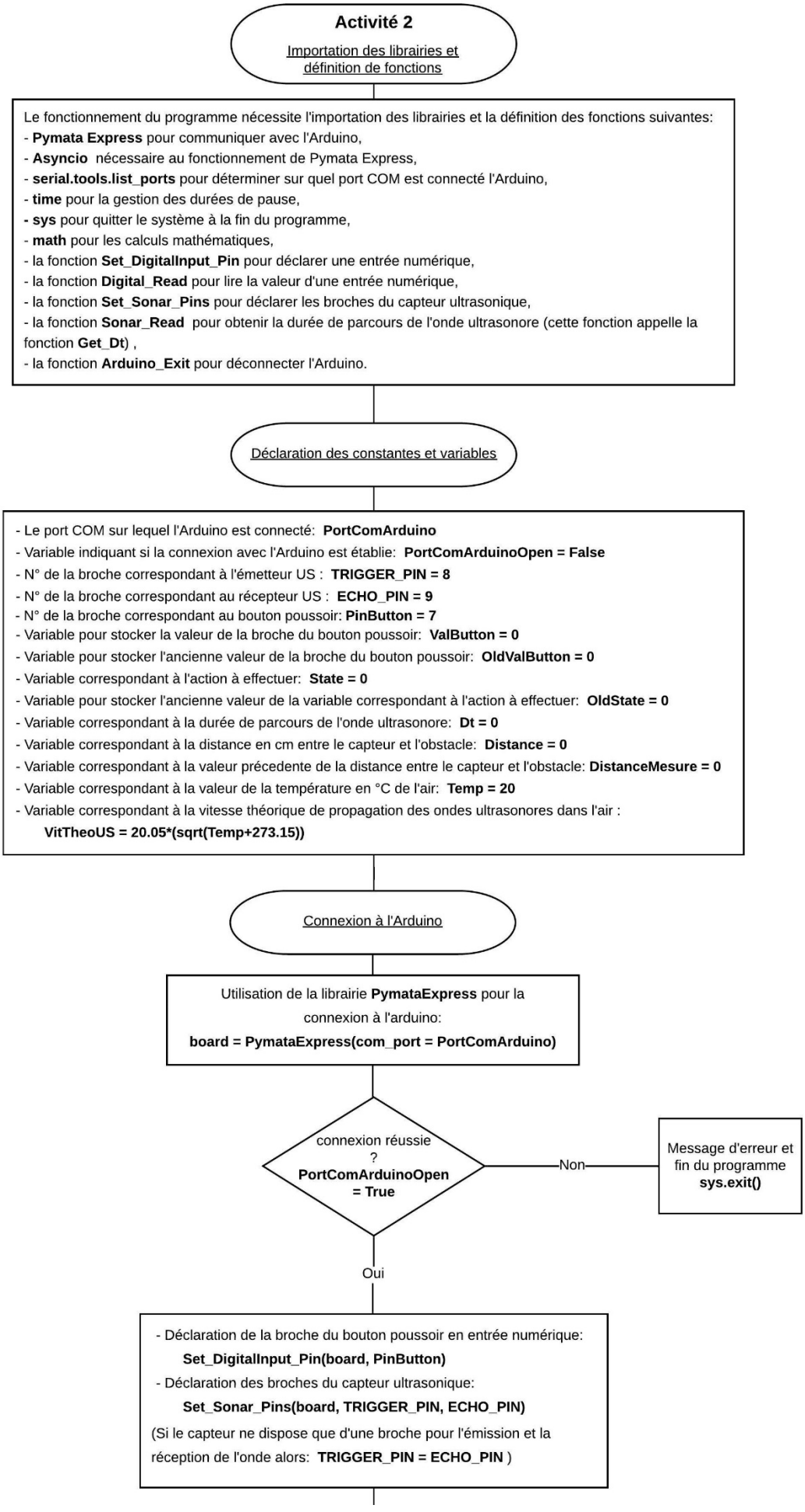
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

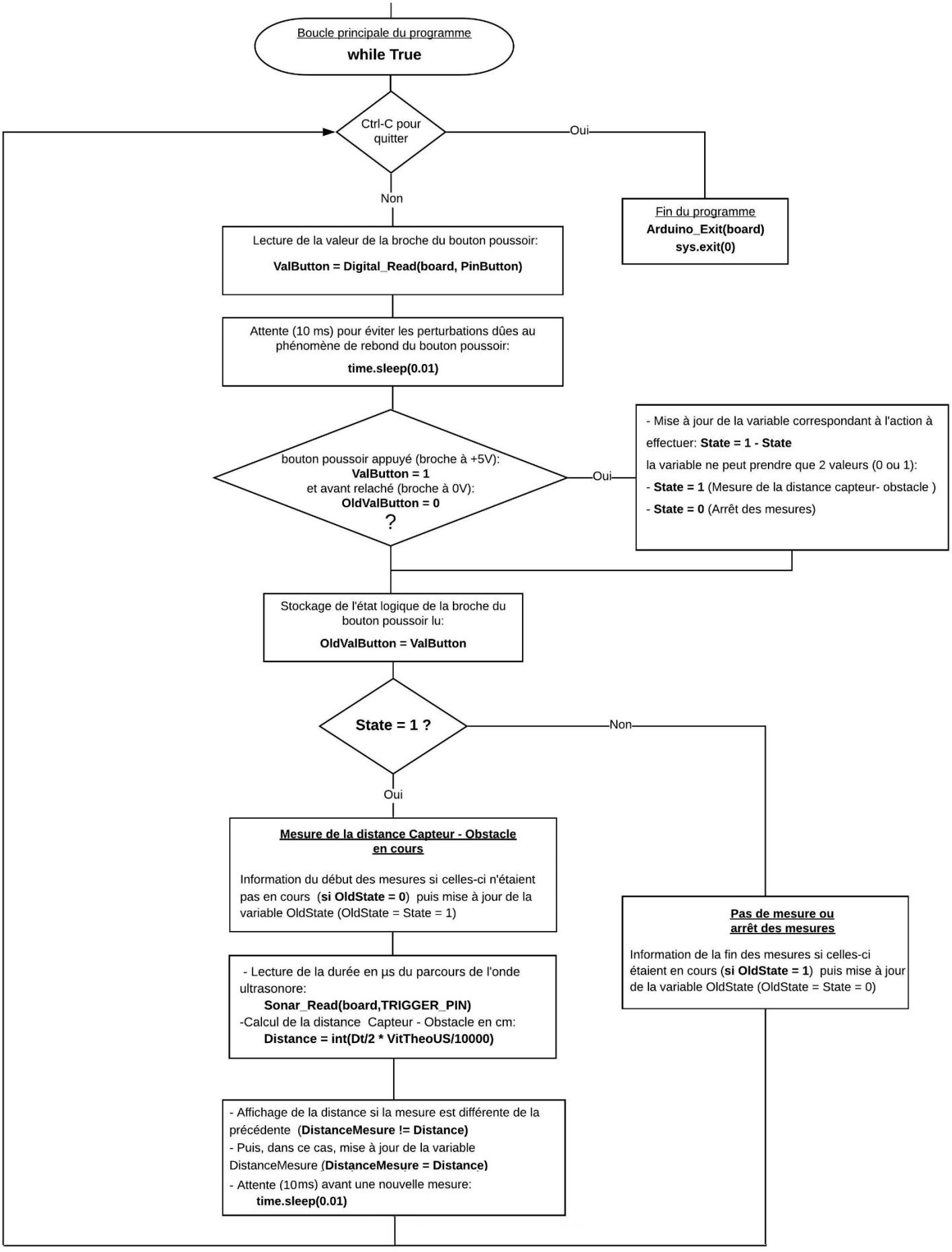
* Algorithme de programmation de l'activité 2 en langage Arduino IDE :




Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Non
Oui
Mesure de la distance en cours
Information dans le moniteur "Série" du début des mesures si celles-ci n'étaient pas en cours (si OldState = 0) puis mise à jour de la variable OldState (OldState = State = 1)
Pas de mesure ou arrêt des mesures
Information dans le moniteur "Série" de la fin des mesures si celles-ci étaient en cours (si OldState = 1) puis mise à jour de la variable OldState (OldState = State = 0)
Capteur à 2 broches (ECHO_PIN != TRIGGER_PIN)
Oui
Non
-Niveau haut pendant 10 µs sur la broche de l'émetteur US:
digitalWrite(TRIGGER_PIN, HIGH)
delayMicroseconds(10)
digitalWrite(TRIGGER_PIN, LOW)
- Mesure de la durée en µs du parcours de l'onde ultrasonore:
Dt = pulseIn(ECHO_PIN, HIGH, MEASURE_TIMEOUT)
L'instruction s'arrête et renvoie 0 si aucune impulsion n'est survenue dans le temps spécifié (MEASURE_TIMEOUT).
-Initialisation de la broche signal du capteur US en sortie et mise à 0 V pendant 2 µs.
pinMode(TRIGGER_PIN, OUTPUT)
digitalWrite(TRIGGER_PIN, LOW)
delayMicroseconds(2)
-Niveau haut pendant 5 µs sur la broche signal du capteur US:
digitalWrite(TRIGGER_PIN, HIGH)
delayMicroseconds(5)
digitalWrite(TRIGGER_PIN,LOW)
-Initialisation de la broche signal du capteur US en entrée :
pinMode(ECHO_PIN,INPUT)
- Mesure de la durée en µs du parcours de l'onde ultrasonore:
Dt = pulseIn(ECHO_PIN, HIGH)
Calcul de la distance en cm
Distance = int(Dt / 2.0 * VitTheoUS/ 10000)
- Affichage de la distance dans le moniteur "Série" si la mesure est différente de la précédente (DistanceMesure != Distance)
-Puis, dans ce cas, mise à jour de la variable DistanceMesure (DistanceMesure = Distance)
Attente (100 ms) avant une nouvelle mesure:
delay(100)

* Algorithme de programmation de l'activité 2 en Python :



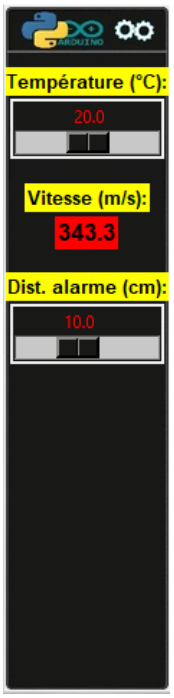

Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Fin du programme
Arduino_Exit(board)
sys.exit(0)
Lecture de la valeur de la broche du bouton poussoir:
ValButton = Digital_Read(board, PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(0.01)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State = 1 - State
la variable ne peut prendre que 2 valeurs (0 ou 1):
- State = 1 (Mesure de la distance capteur- obstacle)
- State = 0 (Arrêt des mesures)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Non
Oui
Mesure de la distance Capteur - Obstacle en cours
Information du début des mesures si celles-ci n'étaient pas en cours (si OldState = 0) puis mise à jour de la variable OldState (OldState = State = 1)
Pas de mesure ou arrêt des mesures
Information de la fin des mesures si celles-ci étaient en cours (si OldState = 1) puis mise à jour de la variable OldState (OldState = State = 0)
- Lecture de la durée en µs du parcours de l'onde ultrasonore:
Sonar_Read(board,TRIGGER_PIN)
-Calcul de la distance Capteur - Obstacle en cm:
Distance = int(Dt/2 * VitTheoUS/10000)
- Affichage de la distance si la mesure est différente de la précédente (DistanceMesure != Distance)
- Puis, dans ce cas, mise à jour de la variable DistanceMesure (DistanceMesure = Distance)
- Attente (10ms) avant une nouvelle mesure:
time.sleep(0.01)

- Activité 3 : Détecteur d'obstacles

Comme il est possible de mesurer une distance avec un Arduino et un capteur à ultrasons, nous allons dans cette activité, réaliser un détecteur d'obstacle qui déclenchera une alarme lumineuse et sonore quand le capteur est situé en dessous d'une distance, d, fixée.

Si le mode de fonctionnement est le "contrôle de l'Arduino", après avoir cliqué sur la prise USB, un menu permettant de régler la température de l'air est affiché. La vitesse théorique du son dans l'air en fonction de la température indiquée est alors calculée et affichée. La distance minimale de déclenchement de l'alarme est également choisie dans ce menu.

Un appui sur le bouton poussoir réel ou virtuel déclenche la mesure de la durée de propagation, **Dt**, de l'onde ultrasonore.
La distance, en cm, entre le capteur et l'obstacle est alors calculée, à partir de **Dt** et de la célérité théorique du son dans l'air, et affichée dans le menu "Résultats".



Menu "Réglages"

Menu "Résultats"

Comme dans l'activité précédente, en mode "simulation", l'utilisateur doit régler la durée de propagation, **Dt**, des ondes ultrasonores dans le menu "Réglages" et après avoir appuyé sur le bouton poussoir virtuel, la distance, en cm, correspondant à ce **Dt** est calculée et affichée le menu "Résultats".

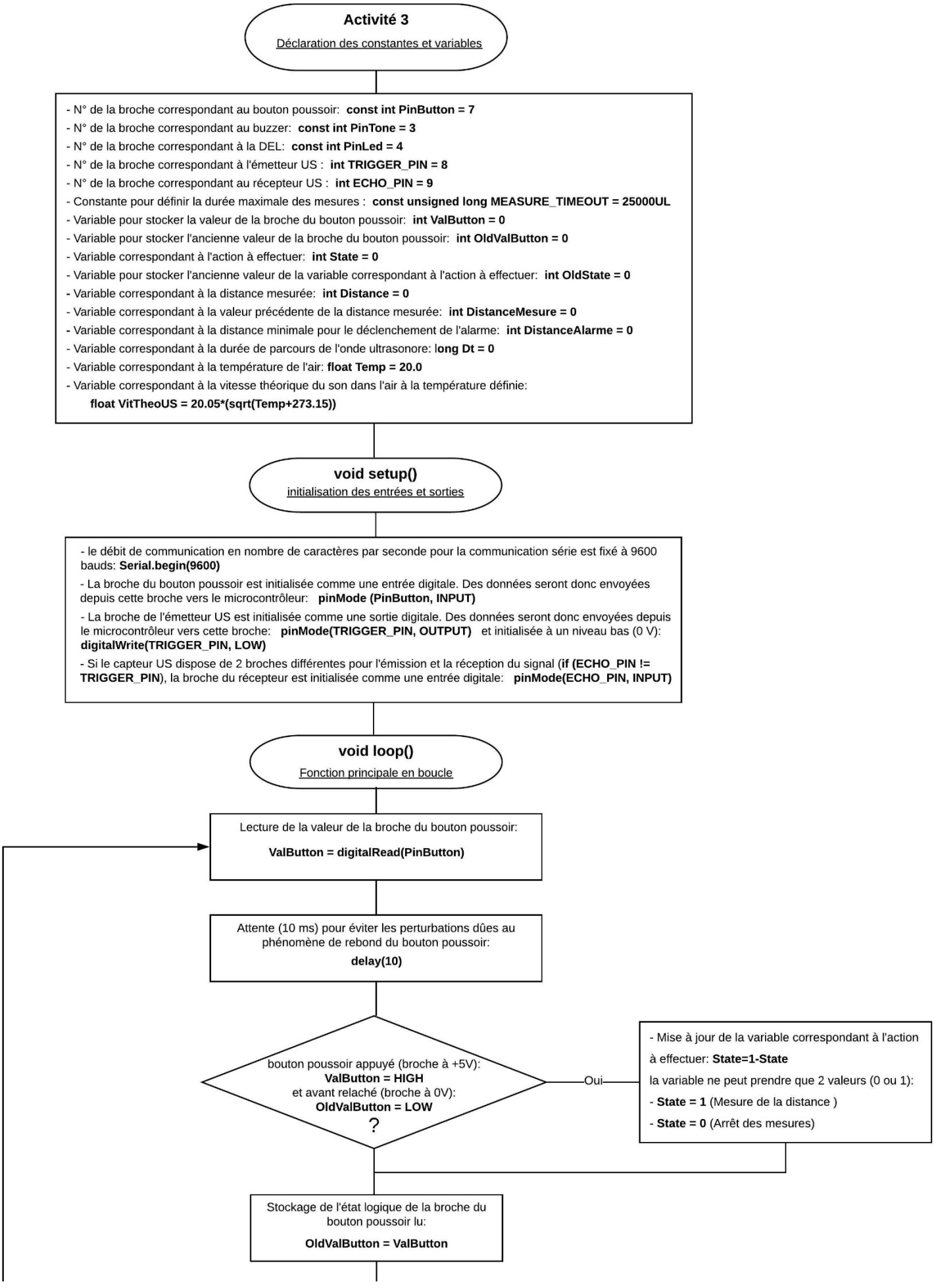
Dans les deux cas, si la distance calculée est inférieure à la distance minimale, l'alarme est déclenchée.

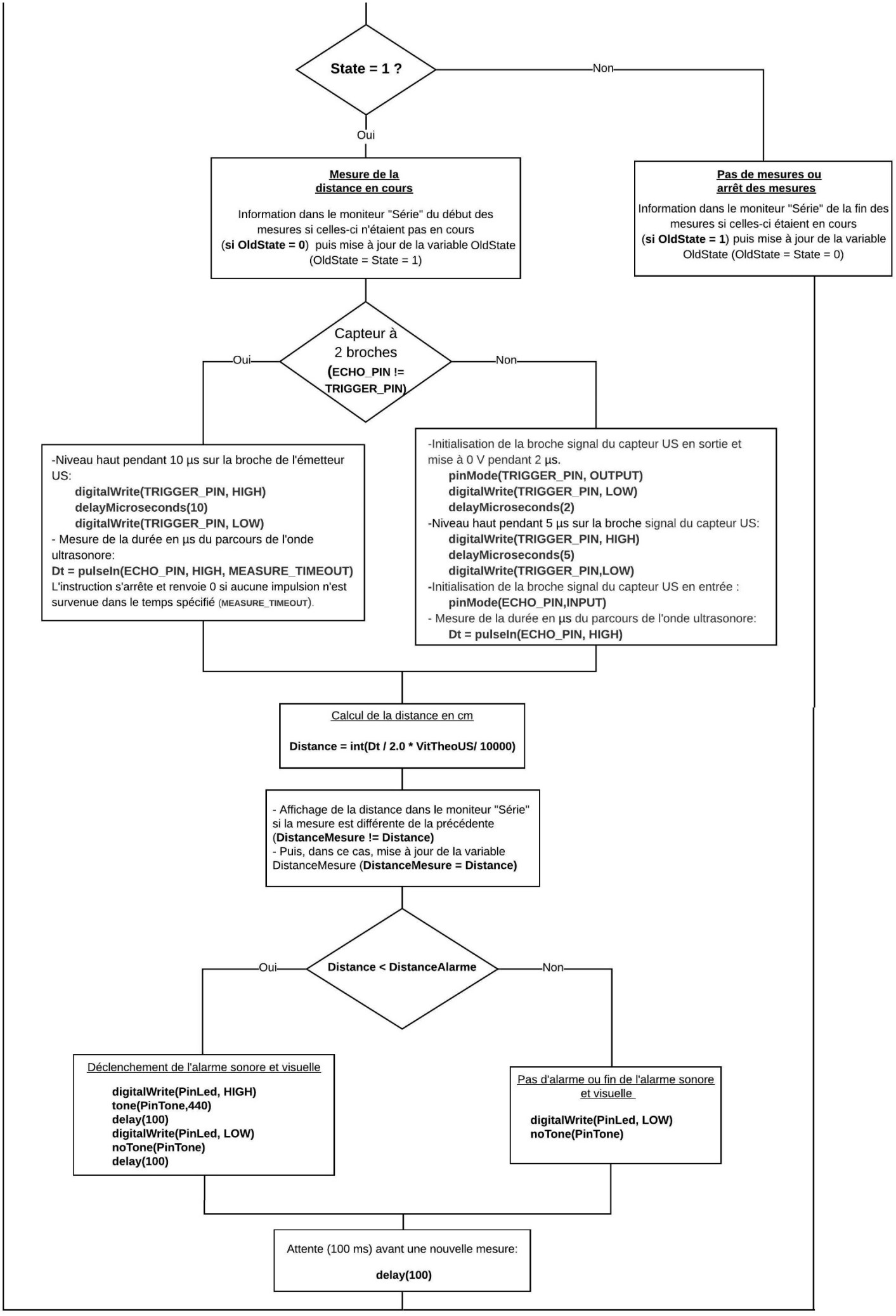
Les mesures sont arrêtées en appuyant de nouveau sur le bouton poussoir réel ou virtuel, en mode "Contrôle de l'Arduino", ou virtuel en mode "Simulation".

A tout moment, il est possible de visualiser le code et son algorithme, programmé en langage Arduino IDE ou en Python, permettant de réaliser cette activité, en cliquant sur les boutons :

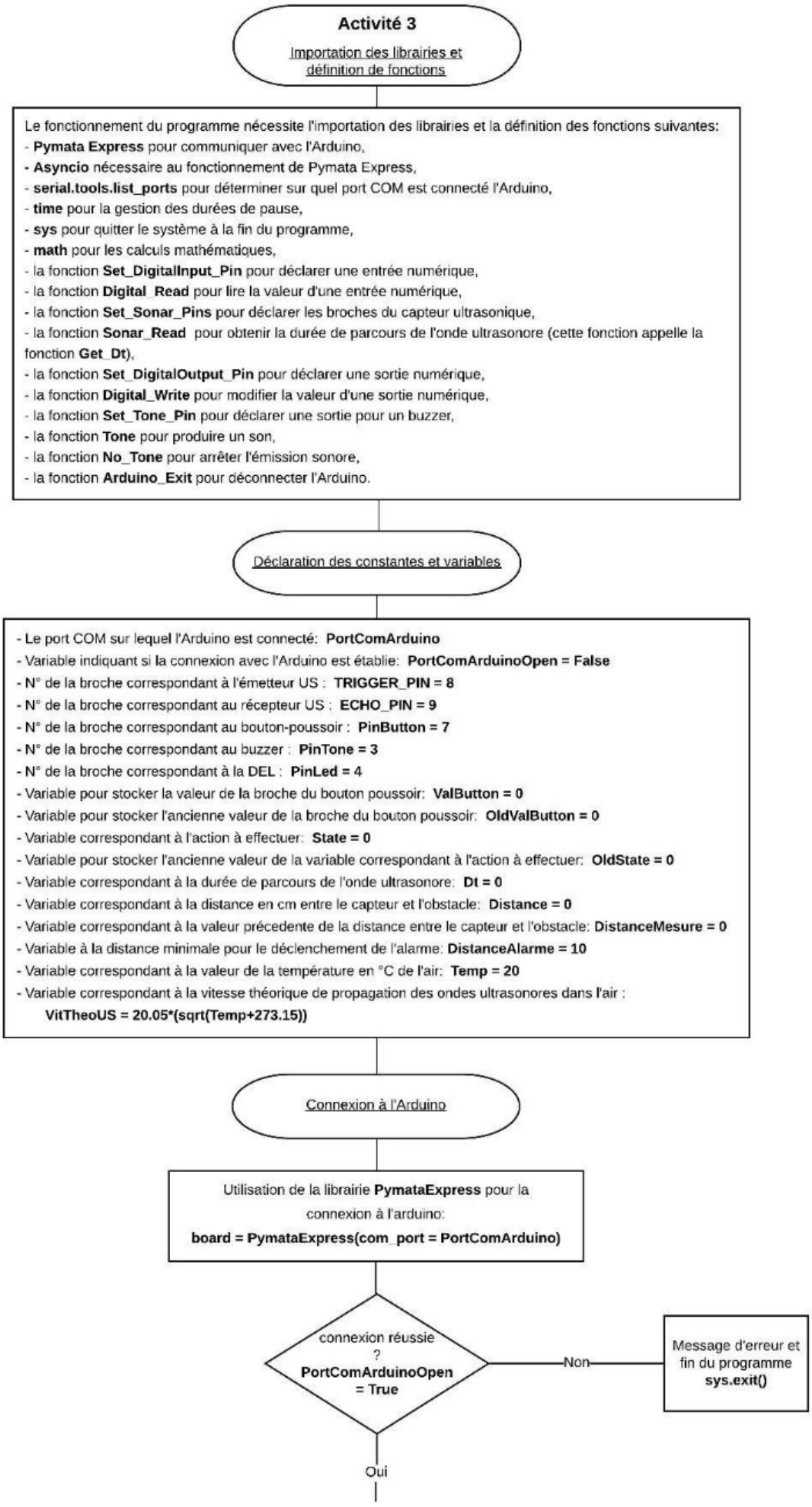
Quel que soit le langage de programmation, les algorithmes du code permettant de réaliser l'activité sont semblables, comme vous pouvez le voir ci-dessous :

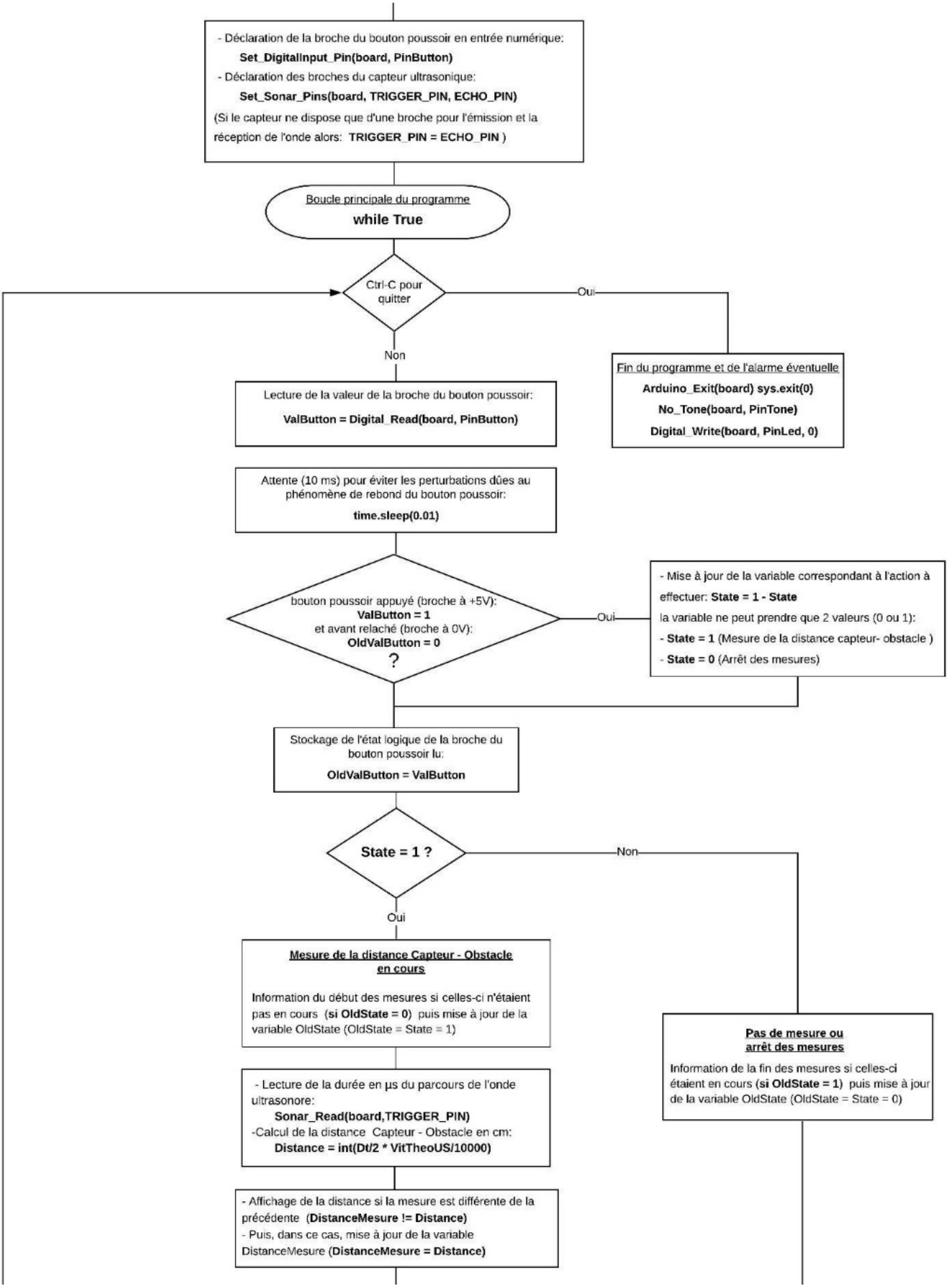
* Algorithme de programmation de l'activité 3 en langage Arduino IDE :




State = 1 ?
Non
Oui
Mesure de la distance en cours
Information dans le moniteur "Série" du début des mesures si celles-ci n'étaient pas en cours (si OldState = 0) puis mise à jour de la variable OldState (OldState = State = 1)
Pas de mesures ou arrêt des mesures
Information dans le moniteur "Série" de la fin des mesures si celles-ci étaient en cours (si OldState = 1) puis mise à jour de la variable OldState (OldState = State = 0)
Capteur à 2 broches (ECHO_PIN != TRIGGER_PIN)
Oui
Non
-Niveau haut pendant 10 µs sur la broche de l'émetteur US:
digitalWrite(TRIGGER_PIN, HIGH)
delayMicroseconds(10)
digitalWrite(TRIGGER_PIN, LOW)
- Mesure de la durée en µs du parcours de l'onde ultrasonore:
Dt = pulseIn(ECHO_PIN, HIGH, MEASURE_TIMEOUT)
L'instruction s'arrête et renvoie 0 si aucune impulsion n'est survenue dans le temps spécifié (MEASURE_TIMEOUT).
-Initialisation de la broche signal du capteur US en sortie et mise à 0 V pendant 2 µs.
pinMode(TRIGGER_PIN, OUTPUT)
digitalWrite(TRIGGER_PIN, LOW)
delayMicroseconds(2)
-Niveau haut pendant 5 µs sur la broche signal du capteur US:
digitalWrite(TRIGGER_PIN, HIGH)
delayMicroseconds(5)
digitalWrite(TRIGGER_PIN,LOW)
-Initialisation de la broche signal du capteur US en entrée :
pinMode(ECHO_PIN,INPUT)
- Mesure de la durée en µs du parcours de l'onde ultrasonore:
Dt = pulseIn(ECHO_PIN, HIGH)
Calcul de la distance en cm
Distance = int(Dt / 2.0 * VitTheoUS/ 10000)
- Affichage de la distance dans le moniteur "Série" si la mesure est différente de la précédente (DistanceMesure != Distance)
- Puis, dans ce cas, mise à jour de la variable DistanceMesure (DistanceMesure = Distance)
Distance < DistanceAlarme
Oui
Non
Déclenchement de l'alarme sonore et visuelle
digitalWrite(PinLed, HIGH)
tone(PinTone,440)
delay(100)
digitalWrite(PinLed, LOW)
noTone(PinTone)
delay(100)
Pas d'alarme ou fin de l'alarme sonore et visuelle
digitalWrite(PinLed, LOW)
noTone(PinTone)
Attente (100 ms) avant une nouvelle mesure:
delay(100)

* Algorithme de programmation de l'activité 3 en Python :




- Déclaration de la broche du bouton poussoir en entrée numérique:
Set_DigitalInput_Pin(board, PinButton)
- Déclaration des broches du capteur ultrasonique:
Set_Sonar_Pins(board, TRIGGER_PIN, ECHO_PIN)
(Si le capteur ne dispose que d'une broche pour l'émission et la réception de l'onde alors: TRIGGER_PIN = ECHO_PIN)
Boucle principale du programme
while True
Ctrl-C pour quitter
Oui
Non
Fin du programme et de l'alarme éventuelle
Arduino_Exit(board) sys.exit(0)
No_Tone(board, PinTone)
Digital_Write(board, PinLed, 0)
Lecture de la valeur de la broche du bouton poussoir:
ValButton = Digital_Read(board, PinButton)
Attente (10 ms) pour éviter les perturbations dûes au phénomène de rebond du bouton poussoir:
time.sleep(0.01)
bouton poussoir appuyé (broche à +5V):
ValButton = 1
et avant relaché (broche à 0V):
OldValButton = 0
?
Oui
- Mise à jour de la variable correspondant à l'action à effectuer: State = 1 - State
la variable ne peut prendre que 2 valeurs (0 ou 1):
- State = 1 (Mesure de la distance capteur- obstacle)
- State = 0 (Arrêt des mesures)
Stockage de l'état logique de la broche du bouton poussoir lu:
OldValButton = ValButton
State = 1 ?
Non
Oui
Mesure de la distance Capteur - Obstacle en cours
Information du début des mesures si celles-ci n'étaient pas en cours (si OldState = 0) puis mise à jour de la variable OldState (OldState = State = 1)
Pas de mesure ou arrêt des mesures
Information de la fin des mesures si celles-ci étaient en cours (si OldState = 1) puis mise à jour de la variable OldState (OldState = State = 0)
- Lecture de la durée en µs du parcours de l'onde ultrasonore:
Sonar_Read(board,TRIGGER_PIN)
-Calcul de la distance Capteur - Obstacle en cm:
Distance = int(Dt/2 * VitTheoUS/10000)
- Affichage de la distance si la mesure est différente de la précédente (DistanceMesure != Distance)
- Puis, dans ce cas, mise à jour de la variable DistanceMesure (DistanceMesure = Distance)


Distance < DistanceAlarme
Oui
Non
Déclenchement de l'alarme sonore et visuelle
Tone(board, PinTone, 440, 0)
Digital_Write(board, PinLed, 1)
time.sleep(0.1)
No_Tone(board, PinTone)
Digital_Write(board, PinLed, 0)
time.sleep(0.1)
Pas d'alarme ou fin de l'alarme sonore et visuelle
No_Tone(board, PinTone)
Digital_Write(board, PinLed, 0)
Attente (10 ms) avant une nouvelle mesure:
time.sleep(0.01)

3. Python - les bases de programmation

3.1. Python, Qu'est-ce que c'est ?

3.2. Installation de Python sous Windows

3.3. Prise en main en mode interactif

3.3.1 Variables et affectation

3.3.2 Les chaines de caractères

3.3.3 Les listes

3.3.4 Les tuples

3.3.5 Les dictionnaires

3.4. Les scripts Python

3.4.1 Structure des scripts Python

3.4.2 Les fonctions

3.4.3 Les fichiers

3.4.4 Les modules – les packages

3.4.5 La programmation orientée objet (classes et objets)

3.1. Python, Qu'est-ce que c'est ?

Python est un langage de programmation interprété.

Il existe plusieurs types de langages. Certains sont compilés, d'autres sont interprétés.

Pour les premiers, une fois que le code est écrit, il faut un compilateur qui le transforme en un langage que seul un ordinateur peut comprendre (le langage machine), en faisant appel au linker qui s'occupe de rassembler les différents fichiers, ainsi qu'au debugger qui vérifie la syntaxe.
On obtient alors un fichier exécutable qui est le programme, avec ses avantages et ses inconvénients :

- le fichier obtenu est optimisé pour l'ordinateur sur lequel il a été compilé, et étant en code machine, ses performances sont très bonnes ;

- chaque fois qu'il faut tester la moindre fonctionnalité du programme, il faut au préalable le recompiler, et, bien sûr, ça ne fonctionnera pas si l'ensemble du code n'est pas opérationnel ;

- enfin, le fichier ne s'exécutera que sur un ordinateur ayant le même système d'exploitation que celui sur lequel il a été compilé.

Les langages interprétés fonctionnent différemment. L'interpréteur interprète directement le code sans qu'il soit obligé de compiler quoi que ce soit. Ce type de langage a également ses avantages et inconvénients :

- gain de temps, possibilité de tester une fonctionnalité en temps réel, etc… ;

- mais, pour faire fonctionner le programme sur un autre ordinateur, il faut que l'interpréteur y soit installé.

Python fait partie de cette seconde catégorie. De plus, l'interpréteur Python peut être installé sous de nombreux systèmes d'exploitation différents : Unix/Linux, Windows, BeOS, Macintosh .... Et bien sûr, vous n'avez pas à adapter votre code, ce sera le même pour tous.

C'est une des raisons pour lesquelles Python a été choisi pour l'enseignement de la programmation au lycée :

- La programmation en lycée s'inscrit dans le prolongement de l'enseignement d'algorithmique, d'informatique et de programmation dispensé au collège en mathématiques et en technologie. Après avoir utilisé un langage de programmation par

blocs (Scratch) au collège, les élèves de lycée doivent utiliser un langage de programmation textuel (Python).

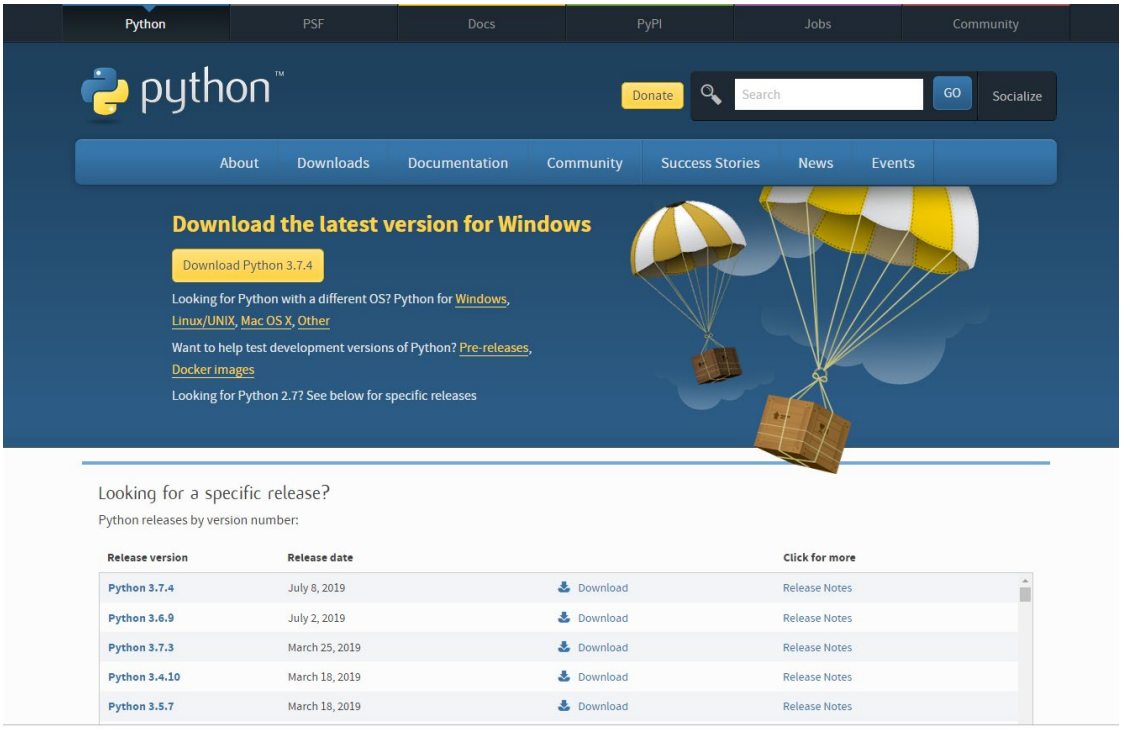
- Jusqu'à la réforme du lycée (rentrée 2019), seuls les mathématiques utilisaient le langage Python.

- Dans les nouveaux programmes, l'utilisation du langage Python au lycée est désormais intégrée à différentes disciplines du lycée général (mathématiques, sciences physiques, chimie, SVT, enseignement scientifique), du lycée technologique (mathématiques, Ingénierie et développement durable) et du lycée professionnel (mathématiques). L'utilisation du Python est également commune au nouvel enseignement général SNT de Seconde, et de spécialité NSI en Première et Terminale.

Le Python devient donc le langage de programmation utilisé par les élèves de lycée. Ce choix traduit une volonté manifeste d'introduire une culture commune autour du codage et d'utiliser un langage simple d'usage, interprété, concis, libre et gratuit, multi-plateforme, largement répandu, riche de bibliothèques adaptées aux thématiques étudiées en classe et bénéficiant d'une vaste communauté d'auteurs dans le monde éducatif.

3.2. Installation de Python sous Windows

. Sur le site www.python.org, cliquez sur « Download » puis choisissez une version de Python 3 en fonction de votre système d'exploitation.

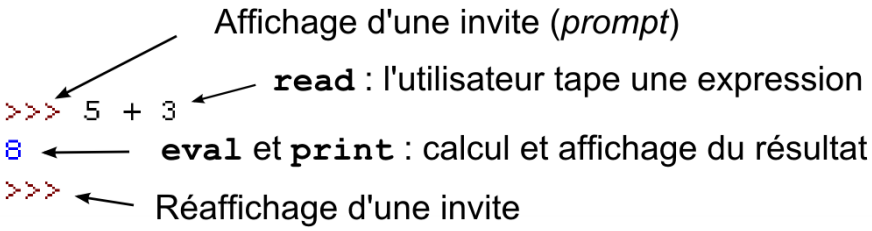


. Exécutez le fichier téléchargé, l'installation se fait ensuite sans aucune difficulté (cliquez sur "Next" à chaque ouverture de fenêtre)

Une fois l'installation effectué, nous allons pouvoir commencer à programmer en Python… Un code python peut être exécuté selon deux modes :

. soit on enregistre un ensemble de commandes Python dans un éditeur (on parle alors d'un script Python) que l'on exécute par une touche du menu de l'éditeur ;

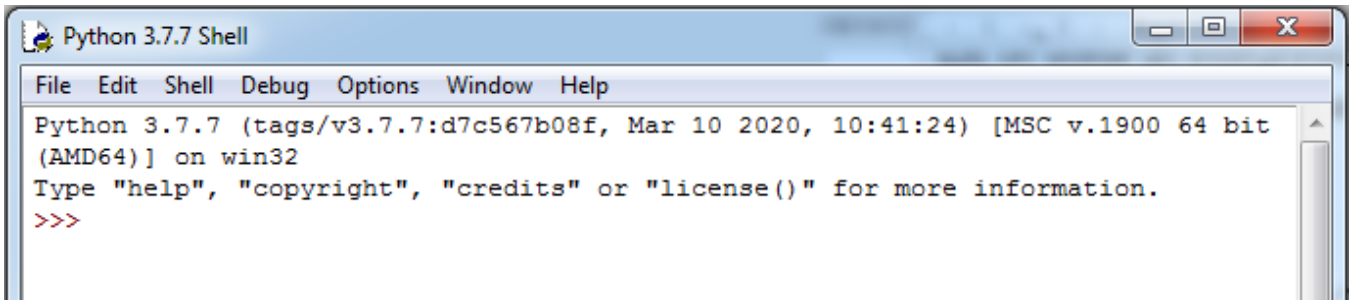
. soit on utilise un interpréteur (par exemple IDLE) pour obtenir un résultat immédiat grâce à l'interpréteur Python embarqué dans IDLE qui exécute la boucle d'évaluation. C'est le mode intéractif.



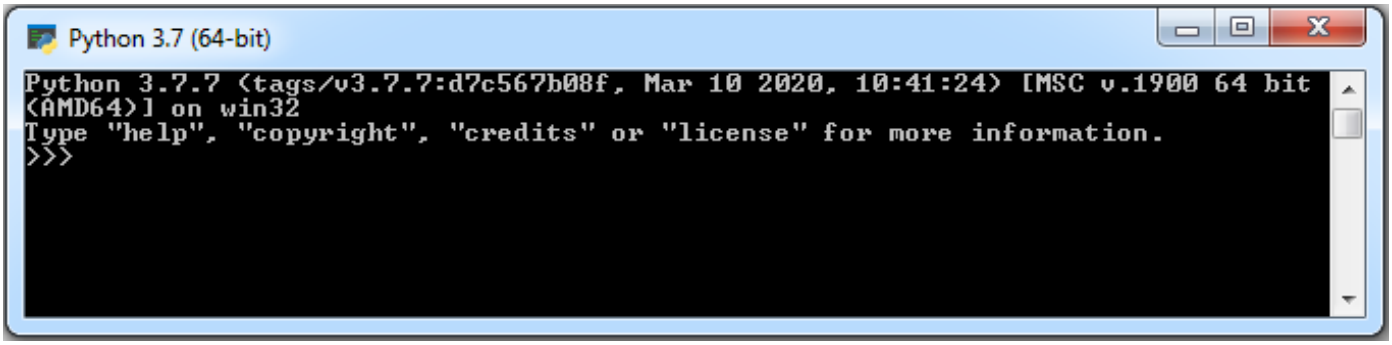
3.3. Prise en main en mode interactif

Pour une utilisation en mode interactif, il suffit de lancer l'interpréteur "**IDLE**" qui se trouve dans le dossier d'installation de Python 3.

Une fenêtre "Python Shell" s'ouvre alors. L'invite de commande >>> signifie que Python est prêt à exécuter une commande.



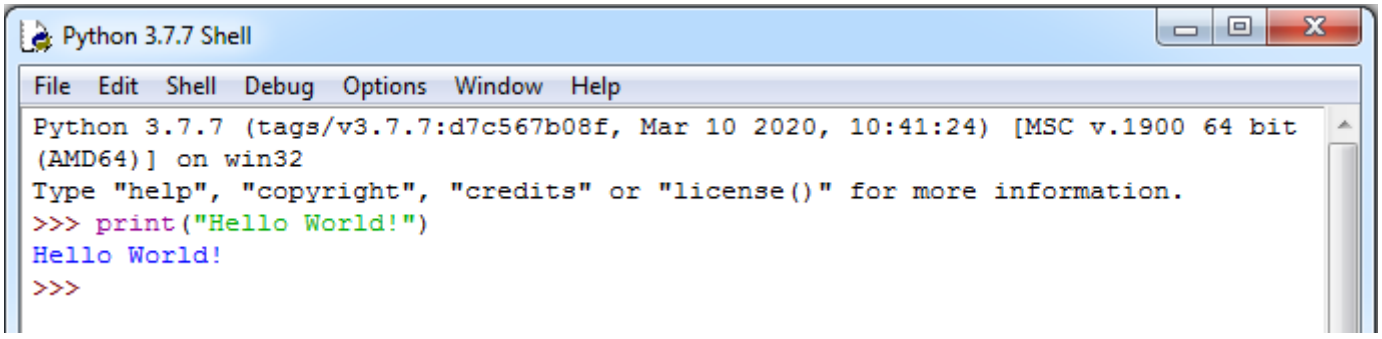
Il est également possible d'utiliser la console Python, également dans le dossier d'installation de Python 3 :



La première commande d'apprentissage de tous les langages de programmation est celle qui affiche la traditionnelle phrase **" Hello World ! "** :

Tapez : **print("Hello World !")**, puis appuyez sur la touche Entrée.

Python exécute cette commande. Le résultat de cette exécution est l'affichage de la chaîne de caractères " Hello World ! ". Une nouvelle invite de commande apparaît alors.



Avec Python, on peut également faire des calculs :

```
================================ RESTART: Shell ================================
>>> 3+2*5
13
>>> 5/2
2.5
>>> 3*(5+1/2)
16.5
>>> 2**5
32
>>> 12%5
2
>>>
```

A noter :

. la virgule des nombres décimaux doit être remplacée par le point.

. 2 exposant 5 s'écrit 2**5

. 12%5 renvoie le reste de la division euclidienne de 12 par 5.

3.3.1. Variables et affectation

Comme tout langage, Python permet de manipuler des données grâce à un vocabulaire de mots réservées et grâce à des types de données. Il utilise des identifiants pour nommer ses objets.

Un identifiant Python valide est une suite non vide de caractères, de longueur quelconque, formée d'un caractère de début et de zéro ou plusieurs caractères de continuation, sachant que :

- un caractère de début peut être n'importe quelle lettre,
- un caractère de continuation est un caractère de début, un chiffre ou un point.

Attention :

Les identifiants sont sensibles à la casse (majuscules ou minuscules) et ne doivent pas être un mot clé réservé de Python tel que :

and; del; from; None; True; as; elif; global; nonlocal; try; assert; else; if; not; while; break; except; import; or; with; class; False; in; pass; yield; continue; finally; is; raise; def; for; lambda; return.

. Les principaux types de données

Il existe différents types de données : le type entier (int), le type nombre à virgule (float), le type Booléen (bool), le type chaîne de caractères (str), …

- Le type int :

Il représente les nombres entiers. Le type **int** n'est limité en taille que par la mémoire de la machine.
Les entiers sont décimaux par défaut, mais on peut aussi utiliser les bases binaire, octale ou hexadécimale.

On peut effectuer les opérations arithmétiques classiques avec des données de type **int :**

```
20 + 3      # 23
20 - 3      # 17
20 * 3      # 60
```

```
20 ** 3       # 8000
20 / 3        # 6.666666666666667
20 // 3       # 6    (division entière)
20 % 3        # 2     (modulo)
abs(3 - 20)   # 17   (valeur absolue)
```

- Le type float :

Un float est un nombre décimal à virgule noté avec un point décimal ou en notation exponentielle :

```
2.718
.02
3e8
6.023e23
```

Les flottants supportent les mêmes opérations que les entiers.

L'import du module math autorise toutes les opérations mathématiques usuelles :

```
import math
print(math.sin(math.pi/4))          # 0.7071067811865475
print(math.degrees(math.pi))        # 180.0
print(math.factorial(9))            # 362880
print(math.log(1024, 2))            # 10.0
```

- Le type bool :

Une donnée de type bool à deux valeurs possibles : **False** et **True**.

Les opérations logiques et de comparaisons sont évaluées afin de donner des résultats booléens False et True :

- Opérateurs de comparaison : **==**, **!=**, **>**, **>=**, **<** et **<=** :

```
2 > 8            # False
2 <= 8 < 15      # True
```

- Opérateurs logiques: **not**, **or** et **and**.

```
(3 == 3) or (9 > 24)        # True
(9 > 24) and (3 == 3)       # False
```

- **Le type str :**

Le type de données **str** représente une séquence de caractères entre guillemets simple ' ou double " ce qui permet d'inclure une notation dans l'autre :

. guillemets = " L'eau vive "
. apostrophes = ' Forme "avec des apostrophes" '

. Les variables

On utilise les variables pour stocker des données. Une variable est un identifiant associé à une valeur. Informatiquement, c'est une référence d'objet située à une adresse mémoire.

On affecte une variable par une valeur en utilisant le signe = (qui n'a rien à voir avec l'égalité en math !). Dans une affectation, le membre de gauche reçoit le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche.

Exemple en mode interactif :

```
>>> pi=3.1415
>>> R=3
>>> Adisc=pi*R**2
>>> Adisc
28.273500000000002
```

. L'exécution de la première ligne crée une variable nommée pi contenant la valeur réelle 3.1415.
. L'exécution de la deuxième ligne crée une variable nommée R contenant la valeur entière 3.
. L'exécution de la troisième ligne crée une variable nommée Adisc contenant le résultat du calcul $pi*R^2$.

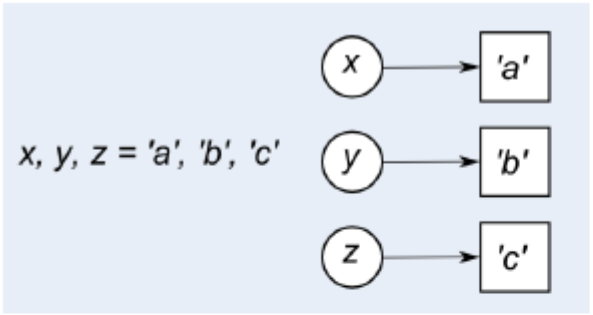
Pour afficher la valeur d'une variable, il suffit de taper son nom puis d'appuyer sur la touche Entrée ou bien taper print(Nom_de_la_variable) :

```
>>> print(Adisc)
28.273500000000002
>>>
```

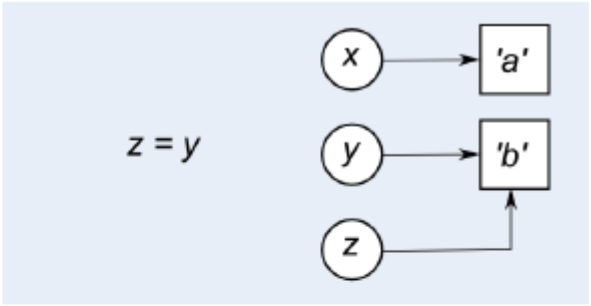
La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps (la valeur antérieure est perdue) :

```
>>> a = 2
>>> a = a + 1
>>> a
3
>>> a = a - 1
>>> a
2
```

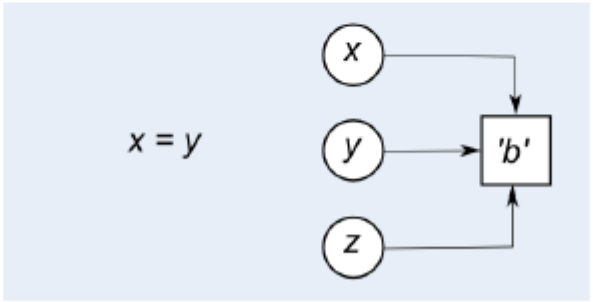
Ceci est résumé dans le schéma suivant, où les cercles représentent les identificateurs (variables) alors que les rectangles représentent les données.
Les affectations relient les identificateurs aux données : si une donnée en mémoire n'est plus reliée, le ramasse-miettes (garbage collector) de Python la supprime automatiquement :



(a) Trois affectations



(b) La donnée 'c' est supprimée



(c) La donnée 'a' est supprimée

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
# affectation simple
v = 4

# affectation augmentée
v += 2 # idem à : v = v + 2 si v est déjà référencé
```

```
# affectation de droite à gauche
c = d = 8 # cibles multiples

# affectations parallèles d'une séquence
e, f = 2.7, 5.1 # tuple
g, h, i = ['G', 'H', 'I'] # liste
x, y = coordonneesSouris() # retour multiple d'une fonction
```

Avec Python, il n'est pas nécessaire de définir préalablement le type de la variable. Le typage se fait automatiquement lors de l'affectation d'une valeur à la variable :

Exemples :

```
>>> a="Hello World !"
>>> b=3
>>> c=2.5
>>> d=[7,3,145]
>>> e=False
>>>
```

- La variable **a** contient une chaîne de caractères, elle sera de type str.

Dès que la valeur d'affection d'une variable est entre guillemets, la variable sera du type « chaîne de caractères » (**str**). Par exemple, si vous saisissez **a="3"** (ou **a='3'**), la variable a est du type chaîne de caractères et la valeur de **a** n'est pas considérée comme un nombre mais comme du texte (Effectuer l'opération a+2 n'aurait aucun sens !) .

- La variable **b** contient un entier, elle sera de type int.

- La variable c contient un nombre à virgule, elle sera de type float.

- La variable d contient une liste, elle sera du type list.

- La variable e contient un booléen, elle sera du type bool (une variable de type bool peut prendre 2 valeurs True ou False).

Pour connaître le type d'une variable il suffit de taper type(nom_de_la_variable) :

```
>>> type(pi)
<class 'float'>
>>>
```

3.3.2. Les chaines de caractères

Les chaines de caractères sont des données de type **str** représentant une séquence de caractères entre guillemets simple ' ' ou double " ".

Trois syntaxes de chaînes sont disponibles :

```
syntaxe1 = "Première forme sans un retour à la ligne"
syntaxe2 = "Deuxième forme avec retour à la ligne\n "
syntaxe3 = """
           Troisième forme multi-ligne
           Troisième forme multi-ligne
           """
```

Ce qui donne dans la console Python:

```
>>> syntaxe1 = "Première forme sans retour à la ligne"
>>> syntaxe2 = "Deuxième forme avec retour à la ligne\n"
>>> print(syntaxe1,syntaxe2,syntaxe1)
Première forme sans retour à la ligne Deuxième forme avec retour à la ligne
 Première forme sans retour à la ligne
>>>
>>> syntaxe3 = """
               Troisième forme multi-ligne
               Troisième forme multi-ligne
                """
>>> print(syntaxe3)

               Troisième forme multi-ligne
               Troisième forme multi-ligne
```

On peut effectuer des opérations sur les chaines :

. Détermination de la longueur de la chaine à l'aide de la fonction **len()**:

```
>>> s = "abcde"
>>> print(len(s))
5
```

. Concaténation de 2 chaines:

```
>>> s1 = "abc"
>>> s2 = "def"
>>> s3 = s1 + s2
>>> print(s3)
abcdef
```

. Répétition d'une chaine :

```
>>> s4 = "A"
>>> s5 = 3*s4
>>> print(s5)
AAA
```

Les chaines sont des objets auxquels on peut appliquer une méthode en utilisant la "notation pointée" entre la donnée/variable à laquelle on applique la méthode et le nom de la méthode : **chaine.methode()**

. Méthodes de test de l'état d'une chaine ch

Les méthodes couramment utilisées suivantes sont à valeur booléennes, c'est-à-dire qu'elles retournent la valeur True ou False.

. **isupper()** et **islower()** retournent True si **ch** ne contient respectivement que des majuscules/minuscules :

```
>>> ch="abcdef"
>>> print(ch.isupper())
False
>>> print(ch.islower())
True
```

. **istitle()** retourne True si seule la première lettre de chaque mot de ch est en majuscule :

```
>>> ch="Abcdef"
>>> print(ch.istitle())
True
```

. **isalnum()**, **isalpha()**, **isdigit()** et **isspace()** retournent True si ch ne contient respectivement que des caractères alphanumériques, alphabétiques, numériques ou des espaces :

```
>>> ch="abcdef"
>>> print(ch.isalnum())
True
>>> ch="abcdef123"
>>> print(ch.isalpha())
False
>>> ch="abcdef"
>>> print(ch.isalpha())
True
```

```
>>> ch="abcdef123"
>>> print(ch.isdigit())
False
>>> ch="abcdef 123"
>>> print(ch.isspace())
False
>>> ch="   "
>>> print(ch.isspace())
True
```

. Méthodes souvent utilisées retournant une nouvelle chaine

. **lower()**, **upper()**, **capitalize()** et **swapcase()** retournent respectivement une chaine en minuscule, en majuscule, en minuscule commençant par une majuscule, ou en casse inversée :

```
>>> ch = "aBcDEf"
>>> print(ch.lower())
abcdef
>>> print(ch.upper())
ABCDEF
>>> print(ch.capitalize())
Abcdef
>>> print(ch.swapcase())
AbCdeF
```

. **strip('chars')**, **lstrip('chars')** et **rstrip('chars')** suppriment toutes les combinaisons de 'chars' (ou l'espace par défaut) respectivement au début et en fin, au début, ou en fin d'une chaine :

```
>>> ch = "AabcdefA"
>>> print(ch.strip('A'))
abcdef
>>> ch = "AabcdAefg"
>>> print(ch.lstrip('A'))
abcdAefg
>>> ch = "AabcdAefgA"
>>> print(ch.rstrip('A'))
AabcdAefg
```

```
>>> ch=" abcdef "
>>> print(ch)
 abcdef
>>> print(ch.lstrip())
abcdef
```

. **split**('sep', maxsplit) découpe la chaine en maxsplit morceaux (tous par défaut) suivant le séparateur 'sep' (ou l'espace par défaut) :

```
>>> ch="azerty1 azerty2 azerty3"
>>> print(ch.split())
['azerty1', 'azerty2', 'azerty3']
>>> print(ch.split(' ',1))
['azerty1', 'azerty2 azerty3']
```

. **rsplit()** effectue la même chose en commençant par la fin :

```
>>> print(ch.rsplit(' ',1))
['azerty1 azerty2', 'azerty3']
```

. **splitlines()** effectue le même travail mais avec les caractères de fin de ligne :

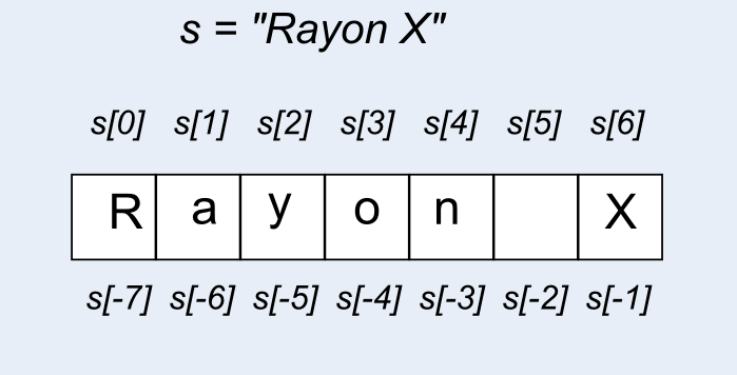
```
>>> ch="azerty1\nazerty2\nazerty3"
>>> print(ch)
azerty1
azerty2
azerty3
>>> print(ch.splitlines())
['azerty1', 'azerty2', 'azerty3']
```

. Indiçage des chaines de caractères

Les chaines de caractères peuvent être indicées avec l'opérateur [ ] dans lequel l'indice, un entier signé qui commence à 0 indique la position d'un caractère :

```
s = "Rayon X"          # len(s) ==> 7
print(s[0])            # R
print(s[2])            # y
print(s[-1])           # X
print(s[-3])           # n
```

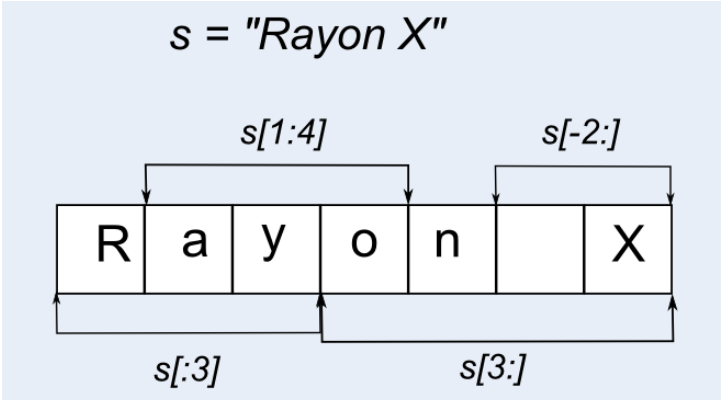
Ci-dessous, un schéma représentatif de l'indiçage de la chaine **s**:



Il est possible d'extraire des sous-chaines de la chaine s :

```
s = "Rayon X"          # len(s) ==> 7
s[1:4]                 # 'ayo' (de l'indice 1 compris à 4 non compris)
s[-2:]                 # ' X' (de l'indice -2 compris à la fin)
s[:3]                  # 'Ray' (du début à 3 non compris)
s[3:]                  # 'on X' (de l'indice 3 compris à la fin)
s[::2]                 # 'RynX' (du début à la fin, de 2 en 2)
```

Opérations d'extraction résumées ci-dessous :



3.3.3 Les listes

En python, les listes sont des variables qui peuvent contenir n'importe quel type de données.
Elles sont notées sous forme d'éléments entre crochets séparés par des virgules.

```
>>> l=['A',2,3,"azerty"]
>>> print(l)
['A', 2, 3, 'azerty']
```

La numérotation des éléments des listes commence à 0.

```
>>> l=['A',2,3,"azerty"]
>>> print(l[0])
A
>>> print(l[2])
3
```

Les listes correspondent à des objets auxquels, il est possible d'appliquer des méthodes :

. **len()** renvoie le nombre d'éléments de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> print(len(l))
4
```

. **append(e)** ajoute un élément **e** à la liste :

```
>>> l.append(5)
>>> print(l)
['A', 2, 3, 'azerty', 5]
```

. **sort()** trie les éléments de la liste si elle contient des données du même type :

```
>>> l=[48,26,75,14]
>>> l.sort()
>>> print(l)
[14, 26, 48, 75]
```

```
>>> l = ['a','j','b','s','azerty']
>>> l.sort()
>>> print (l)
['a', 'azerty', 'b', 'j', 's']
```

. **remove(e)** retire l'élément e de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> l.remove(2)
>>> print(l)
['A', 3, 'azerty']
```

. **pop()** enlève le dernier élément de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> dernier_element=l.pop()
>>> print(l)
['A', 2, 3]
>>> print(dernier_element)
azerty
```

. **pop(i)** enlève l'élément d'indice **i** de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> element_1=l.pop(1)
>>> print(l)
['A', 3, 'azerty']
>>> print(element_1)
2
```

. **index(e)** retourne la position de l'élément **e** de la liste **:**

```
>>> l=['A',2,3,"azerty"]
>>> print(l.index(2))
1
```

. **reverse()** inverse l'ordre des éléments de la liste :

```
>>> l=['A',2,3,"azerty"]
>>> l.reverse()
>>> print(l)
['azerty', 3, 2, 'A']
```

. **count(e)** compte le nombre d'occurrence de l'élément e dans la liste :

```
>>> l=[1,5,8,5,6,4]
>>> print(l.count(2))
0
>>> print(l.count(5))
2
```

. **extend()** concatène deux listes :

```
>>> l1=['A',2,3,"azerty"]
>>> l2=['B',4,5]
>>> l1.extend(l2)
>>> print(l1)
['A', 2, 3, 'azerty', 'B', 4, 5]
```

Remarques :

. Une liste **l** vide s'écrit : **l = []**

. La fonction **del** permet de supprimer un élément d'index **i** d'une liste :

```
>>> l=['A',2,3,"azerty"]
>>> del l[0]
>>> print(l)
[2, 3, 'azerty']
```

. Les expressions d'indiçage des chaines de caractères s'appliquent aussi aux listes :

```
>>> l=['A',2,3,"azerty"]
>>> print(l[1:2])
[2]
>>> print(l[1:3])
[2, 3]
>>> print(l[-1])
azerty
>>> print(l[:-2])
['A', 2]
>>> print(l[-2:])
[3, 'azerty']
```

. La méthode **split()** permet de transformer une chaine de caractère en liste :

```
>>> s="je veux couper la chaine"
>>> l=s.split(' ')
>>> print(l)
['je', 'veux', 'couper', 'la', 'chaine']
```

. La méthode **join()** permet de transformer une liste de chaine en une chaine de caractère :

```
>>> s=' '.join(l)
>>> print(s)
je veux couper la chaine
```

. En plus de la méthode **count()**, on peut également savoir si un élément est dans une liste, en utilisant le mot clé **in** de cette manière:

```
>>> liste = [1,2,3,5,10]
>>> 3 in liste
True
>>> 6 in liste
False
```

. A la place de la méthode **extend()** , on peut additionner deux listes pour les combiner ensemble en utilisant l'opérateur **+** :

```
>>> l1=['A',2,3,"azerty"]
>>> l2=['B',4,5]
>>> l3=l1+l2
>>> print(l3)
['A', 2, 3, 'azerty', 'B', 4, 5]
```

. Il est également possible de multiplier des listes :

```
>>> l=['a','b']
>>> l=l*5
>>> print(l)
['a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'a', 'b']
```

ce qui est pratique pour initialiser une liste :

```
>>> l=[0]*5
>>> print(l)
[0, 0, 0, 0, 0]
```

. La fonction **range()** génère une liste composée d'une simple suite arithmétique :

```
>>> liste = range(5)
>>> print(liste)
range(0, 5)
```

- La fonction **list()** crée une liste (à partir d'une chaine, d'un tuple ou d'une liste) :

```
>>> list(liste)
[0, 1, 2, 3, 4]
>>> l=['A',2,3,"azerty"]
>>> list(l)
['A', 2, 3, 'azerty']
```

- On peut préciser l'entier de départ (**range(entier de départ inclus, entier de fin exclu)**) :

```
>>> liste = range(3,10)
>>> list(liste)
[3, 4, 5, 6, 7, 8, 9]
```

- Et l'incrément (**range(entier de départ inclus, entier de fin exclu, incrément)**) :

```
>>> liste = range(2,10,2)
>>> list(liste)
[2, 4, 6, 8]
```

. Pour afficher les éléments d'une liste on peut aussi utiliser une boucle **For** :

```
>>> liste = range(5)
>>> print(liste)
range(0, 5)
>>> for elt in liste: print(elt)

0
1
2
3
4
```

. La fonction **enumerate**() permet en plus de récupérer l'index de l'élément :

```
>>> liste=['A',2,3,"azerty"]
>>> for elt in enumerate(liste): print(elt)

(0, 'A')
(1, 2)
(2, 3)
(3, 'azerty')
```

Les valeurs retournées par la boucle sont des tuples.

3.3.4 Les tuples

Les tuples sont des listes qui ne peuvent pas être modifiées. Ils sont notés sous forme d'éléments entre parenthèses séparés par des virgules.

```
>>> t = ("un", 2, "trois")
>>> print(t)
('un', 2, 'trois')
```

Les parenthèses ne sont pas obligatoires :

```
>>> t = 1,2,3
>>> type(t)
<class 'tuple'>
```

Le tuple étant une sorte de liste, on peut donc utiliser la même syntaxe pour lire les données du tuple.

```
('un', 2, 'trois')
>>> print(t[1])
2
```

Mais si on essaie de changer la valeur d'un index, l'interpréteur affiche un message d'erreur :

```
>>> t[1]='deux'
Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <module>
    t[1]='deux'
TypeError: 'tuple' object does not support item assignment
```

Cependant, le tuple permet une affectation multiple :

```
>>> t = (1,2)
>>> var1,var2 = t
>>> print(var1)
1
>>> print(var2)
2
```

Ou

```
>>> var3,var4 = 3,4
>>> print (var3, var4)
3 4
```

Il sera donc principalement utilisé pour définir les constantes des programmes.

3.3.5 Les dictionnaires

Comme les listes, les dictionnaires permettent de stocker des données mais au lieu d'utiliser des index pour les repérer, on utilise des clés alphanumériques.

Chaque élément d'un dictionnaire est composé de 2 parties, on parle de paires "clé/valeur".

- Pour ajouter des données à un dictionnaire il faut donc indiquer une clé ainsi qu'une valeur :

. Création d'un dictionnaire :

dictionnaire = {clé1:valeur1, clé2:valeur2,…}

dictionnaire = {} (dictionnaire vide)

ou

dictionnaire = dict([(clé1,valeur1), (clé2:valeur2),…]) (liste de tuples)

dictionnaire = dict() (dictionnaire vide)

. Ajout d'une donnée :

```
>>> inventaire = {}
>>> inventaire["bécher"]= 10
>>> print(inventaire)
{'bécher': 10}
>>> inventaire["éprouvette"]= 20
>>> print(inventaire)
{'bécher': 10, 'éprouvette': 20}
```

- La méthode **get ()** permet de récupérer une valeur du dictionnaire :

```
>>> print(inventaire.get('bécher'))
10
```

- pour effacer une entrée (clé/valeur), on utilise la fonction **del** :

```
>>> del inventaire["bécher"]
>>> inventaire
{'éprouvette': 20}
```

- A l'aide d'une boucle **for** et de la méthode **keys()**, on peut récupérer les clés d'un dictionnaire :

```
>>> inventaire={"bécher":10,"éprouvette":20}
>>> for cle in inventaire.keys(): print(cle)

bécher
éprouvette
```

- et avec la méthode **values()**, on récupère les valeurs :

```
>>> for valeur in inventaire.values(): print(valeur)

10
20
```

- et pour récupérer les clés et les valeurs en même temps, on utilise la méthode **items()** qui retourne un tuple :

```
>>> inventaire=dict([("bécher",10),("éprouvette",20),("erlenmeyer",15)])
>>> for cle,valeur in inventaire.items():
        print (cle, valeur)


bécher 10
éprouvette 20
erlenmeyer 15
```

3.4. Les scripts Python

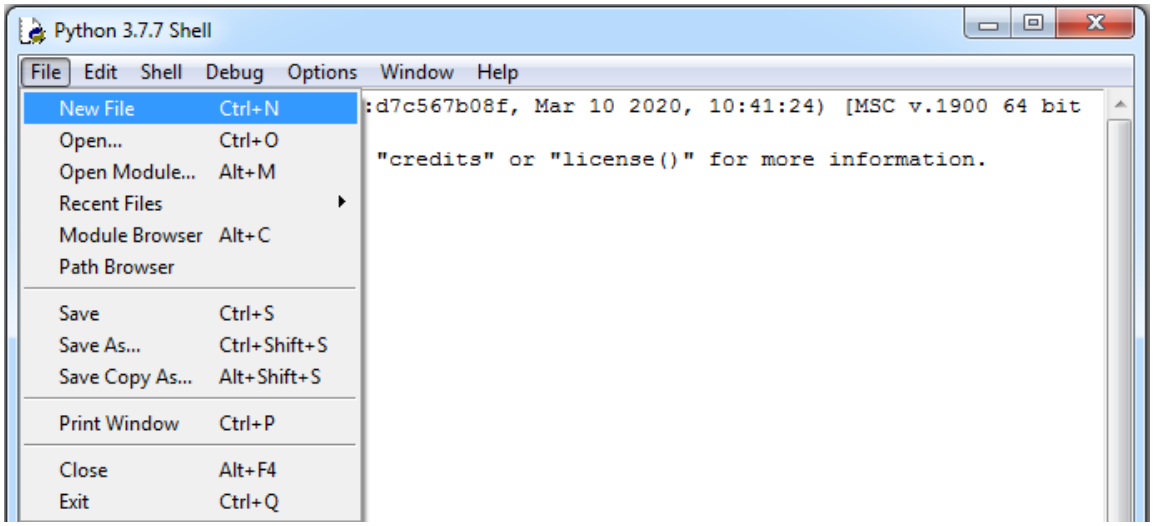
En mode interactif, les lignes d'instructions ne sont plus accessibles une fois exécutées. Mais il est bien-sûr possible d'écrire et de conserver un programme (un script), à l'aide d'un éditeur, pour pouvoir l'exécuter à loisir ou pour le modifier ultérieurement.

Il existe de nombreux éditeurs de scripts Python qui intègre également un interpréteur pour exécuter les programmes. C'est ce qu'on appelle un IDE ou Environnement de développement (Integrated Development Environment).

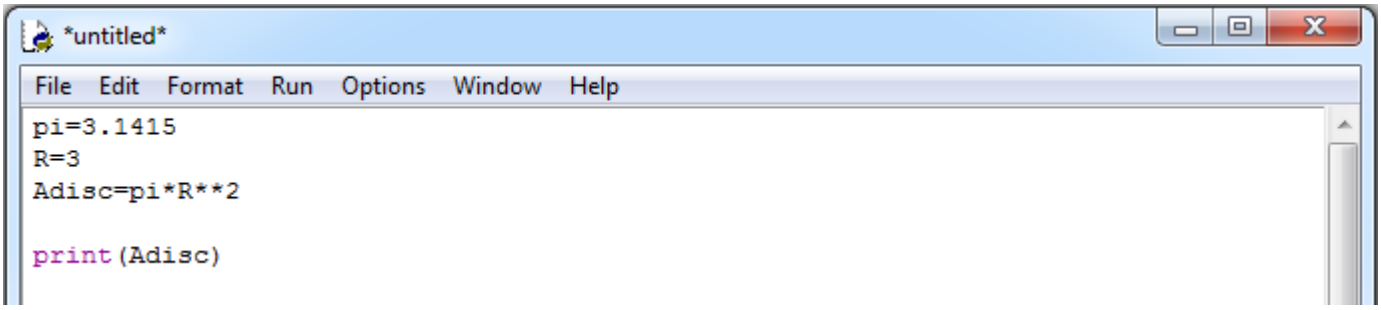
C'est un environnement de programmation complet qui se présente sous la forme d'une application. Il se compose généralement d'un éditeur de code, d'un interpréteur, d'un débogueur... On peut citer **Pycharm**, **Spider**…

Mais pour une initiation à la programmation en Python, l'utilisation de l'interpréteur **IDLE** qui permet également d'éditer des scripts est tout à fait adapter :

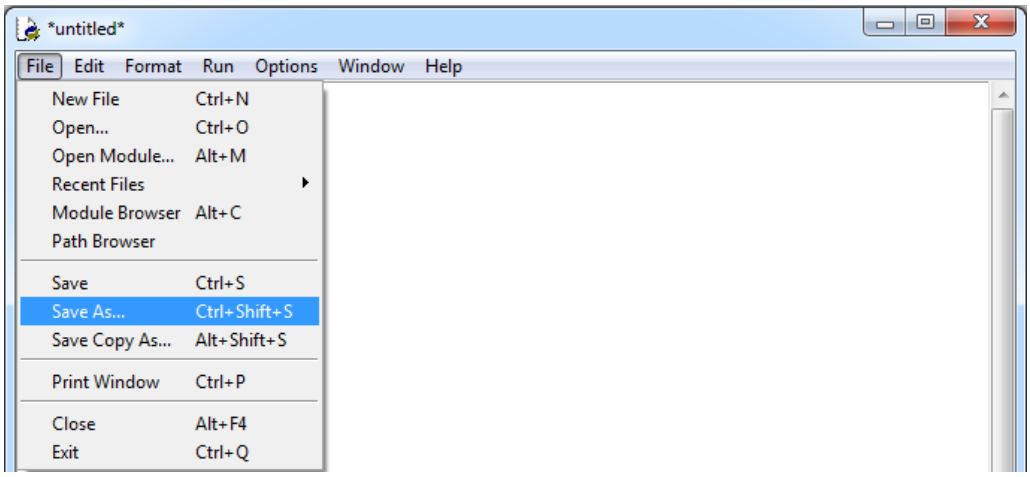
- Dans la fenêtre Python Shell (celle du mode interactif), sélectionnez "**New File**" dans le menu "**File**" :



- Une nouvelle fenêtre s'ouvre alors. C'est dans cette fenêtre que l'on va écrire notre premier programme (calcul de la surface d'un disque de rayon R) :



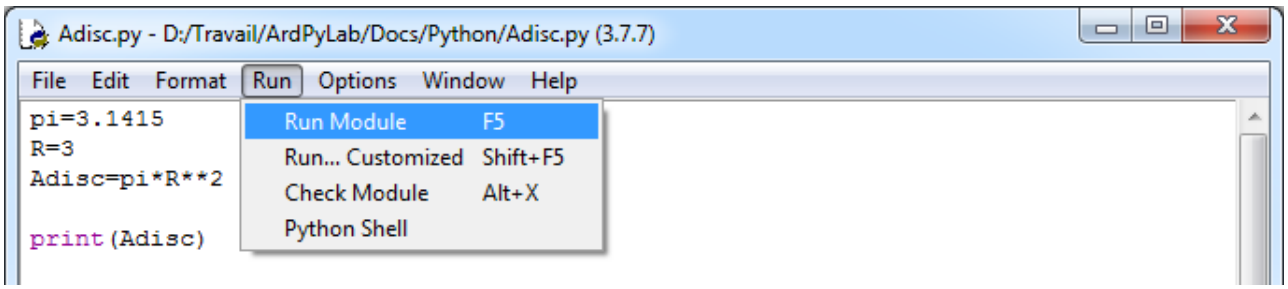
- Enregistrement du programme : Sélectionnez ”**Save as”** dans le menu ”**File”** :



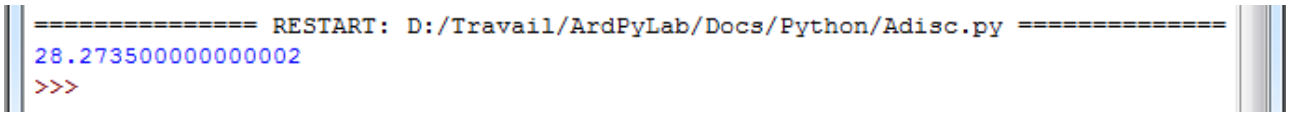
Une fenêtre d’enregistrement s’ouvre alors. Choisissez dans l’arborescence le dossier dans lequel vous voulez enregistrer le programme, puis dans le champ d’enregistrement du fichier saisissez le nom du programme suivi de l’extension ”.py”, puis cliquez sur enregistrer.

- Pour ouvrir un programme python précédemment enregistré, il suffira de sélectionner ”**Open**” dans le menu ”**File**” de l’environnement IDLE puis de chercher dans vos dossiers le fichier python à ouvrir.

- Exécution du programme : Pour exécuter le programme, il suffit de sélectionner ”**Run Module**” dans le menu ”**Run**” (si une modification du script a été effectuée, on vous proposera d’enregistrer le script modifié avant de l’exécuter).



Le programme s’exécute dans la fenêtre Python shell :



On pourra alors modifier notre premier programme, en demandant par exemple à l’utilisateur de saisir le rayon du disque. Les modifications seront enregistrées et le programme ré-exécuté autant de fois que vous souhaitez.

3.4.1 Structure des scripts Python

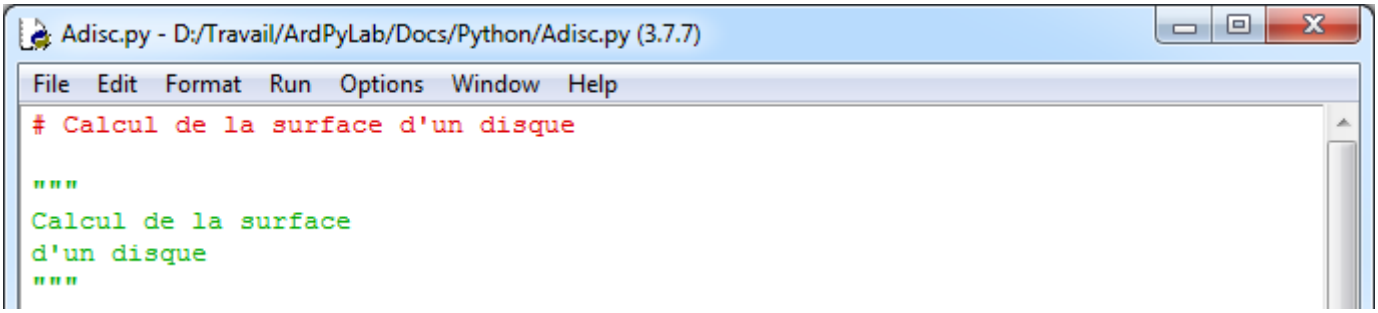
Un script Python est formée d'une suite d'instructions exécutées de haut en bas du script.

. Les instructions

Chaque instruction s'écrit sur une ligne, il n'y a pas de séparateur d'instruction. Si une ligne est trop grande, le caractère "****" permet de passer à la ligne suivante.

On utilise le caractère "**#** "pour insérer des commentaires dans un programme. Les commentaires vont du caractère "**#**" jusqu'à la fin de la ligne.

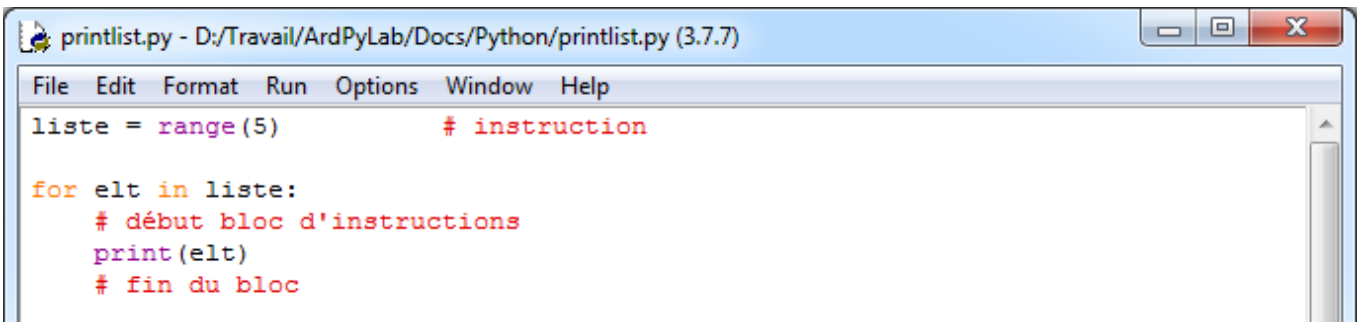
Il n'existe pas de commentaires en bloc comme en C **(/* ... */)**. Mais il est possible d'utiliser des triples guillemets doubles ou simples (**'''**) avant et après le bloc de commentaires comme pour déclarer une variable chaine de caractères sur plusieurs lignes. Le bloc n'étant pas rattaché à une variable, il sera ignoré par l'interpréteur.
Sous IDLE, le plus simple est cependant de sélectionner le bloc de commentaires et de le déclarer comme tel avec "**Format/comment out region**".



. Les blocs d'instructions

Les blocs d'instructions sont matérialisés par des indentations (plus de **{** et **}** comme en C !).

Le caractère "**:**" sert à introduire les blocs.



. Les entrées-sorties

L'utilisateur a généralement besoin d'interagir avec le programme. En l'absence d'interface graphique, en mode "console" (Python shell), on doit pouvoir saisir ou entrer des informations, ce qui est fait depuis une lecture au clavier. Inversement, on doit pouvoir afficher ou sortir des informations, ce qui correspond généralement à une écriture sur l'écran (dans la console).

- Les entrées

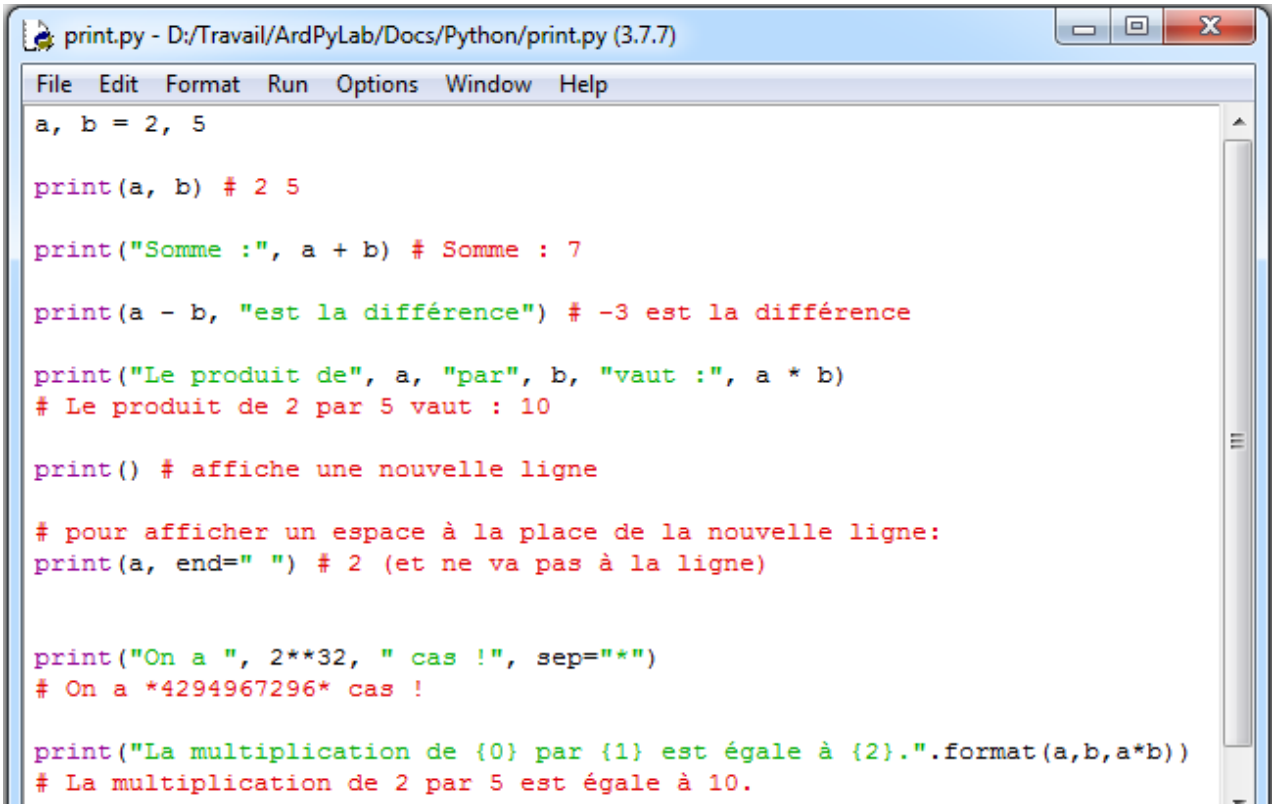
Il s'agit de réaliser une saisie dans la console Python. Pour cela, on utilise la fonction "**input()**" qui interrompt le programme, affiche une éventuelle invite et attend que l'utilisateur entre une donnée et la valide par la touche "**Entrée** " .

La fonction "**input()**" effectue toujours une saisie en mode texte (la saisie est une chaine – type '**str**') dont on peut ensuite changer le type par une conversion.

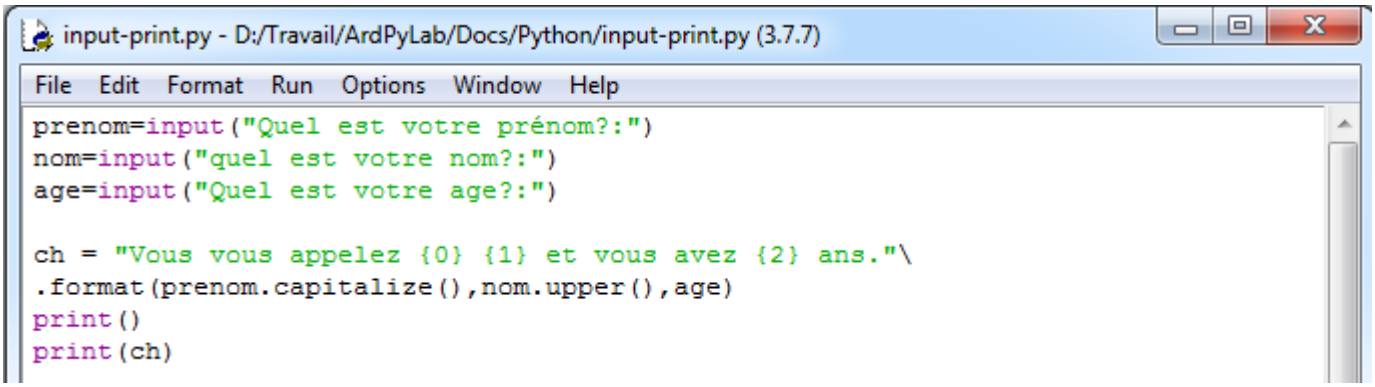
```
>>> nb = input("veuillez saisir un nombre:")
veuillez saisir un nombre:45
>>> print(type(nb))
<class 'str'>
>>> ch = input("veuillez saisir une chaine de caractères:")
veuillez saisir une chaine de caractères:azert123
>>> print(type(ch))
<class 'str'>
```

- Les sorties

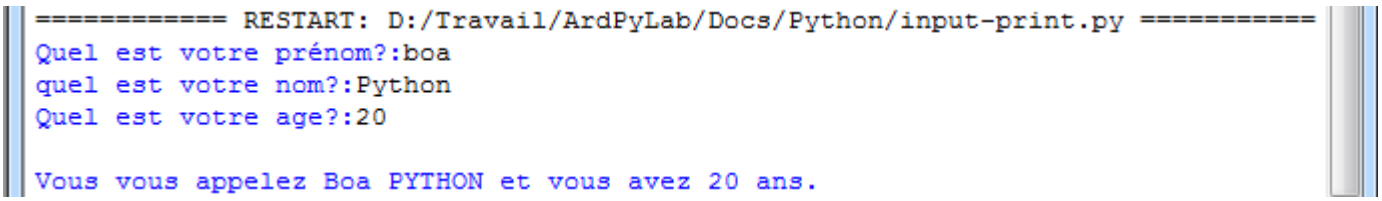
En mode interactif, Python lit, évalue et affiche, mais la fonction "**print()**" reste indispensable aux affichages dans les scripts :



Exemple : Le programme ci-dessous demande le nom, le prénom et l'age de l'utilisateur et affiche les données après formatage.



Résultat dans la fenêtre Python shell :



. Les conversions

Il existe un certain nombre de fonctions permettant de convertir les données d'un type à l'autre.
La fonction "**type()**" permet de récupérer le type de la donnée sous forme d'une chaîne.

Fonction	Description	Exemple
ord	retourne la valeur ASCII d'un caractère	ord('A')
chr	retourne le caractère à partir de sa valeur ASCII	char(65)
str	convertit en chaîne	str(10), str([10,20])
int	interprète la chaîne en entier	int('45')
int long	interprète la chaîne en entier long	long('56857657695476')
float	interprète la chaîne en flottant	float('23.56')

Autre conversions :

. **Conversion binaire**

La fonction "**bin()**" permet de convertir un nombre binaire en chaine de caractères :

```
>>> bin(204)
'0b11001100'
```

Pour convertir une chaine de caractères représentant un nombre binaire (base 2), on utilise la fonction **int()** en précisant la base :

```
>>> int('11001100',2)
204
```

Le préfixe 0b n'est pas obligatoire et peut être supprimé.

. **Conversion hexadécimale**

La fonction "**hex()**" permet de convertir un nombre hexadécimal en chaine de caractères et la fonction **int()** en précisant la base 16 fait la conversion inverse:

```
>>> hex(32)
'0x20'
>>> int('0x20',16)
32
```

. Les structures de contrôles

- Condition **if** :

L'instruction **if** ("si" en français), utilisée avec un opérateur logique de comparaison, permet de tester si une condition est vraie.

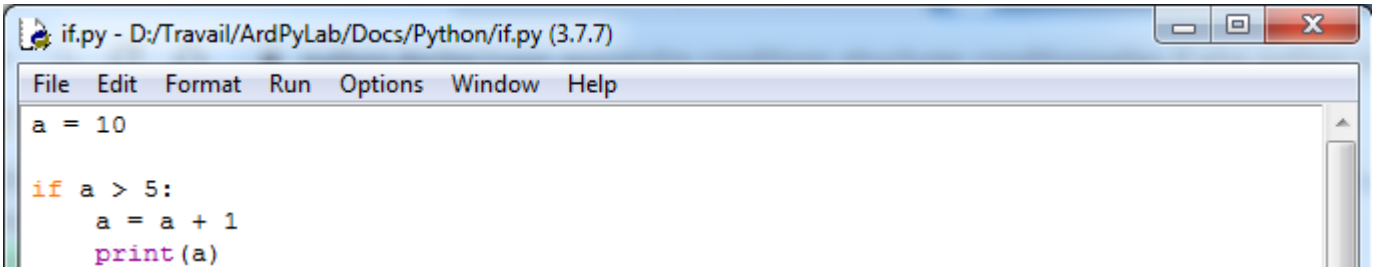
Le format d'un test **if** est le suivant :

```
if  uneVariable > 50 :
    # Instructions  (attention aux tabulations !)
```

Dans cet exemple, le programme va tester si la variable **uneVariable** est supérieure à 50. Si c'est le cas, le programme va réaliser une action particulière. Autrement dit, si l'état du test est vrai, le bloc d'instructions (ligne d'instructions de même indentation) après le caractère "**:**" est exécuté.

Exemple :

Une valeur est donnée à une variable et si cette valeur est supérieure à 5, alors celle-ci est incrémentée de 1 et affichée.



Résultat dans la fenêtre Python Shell :

```
================ RESTART: D:/Travail/ArdPyLab/Docs/Python/if.py ================
11
>>>
```

Rappel des opérateurs logiques de comparaison :

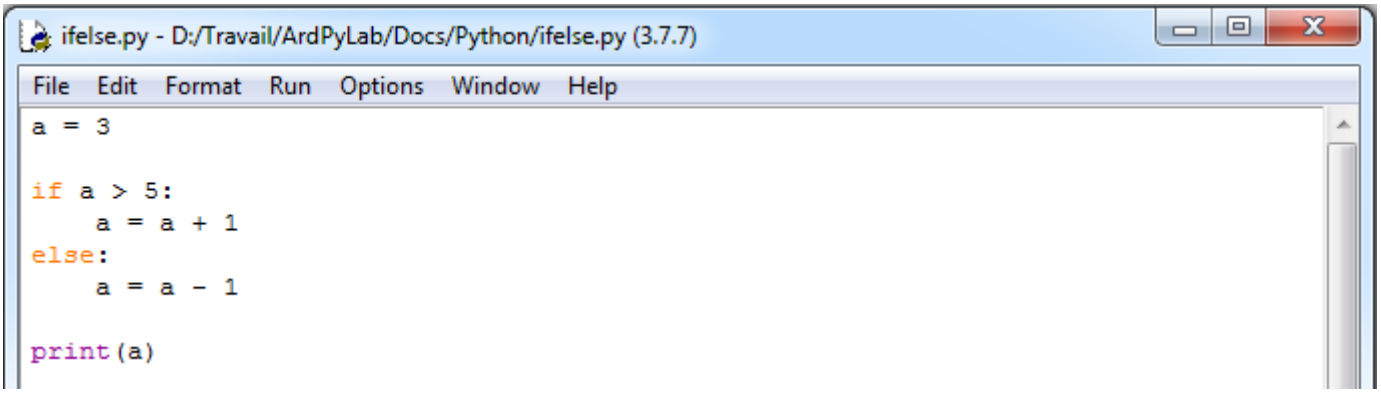
- x == y est vrai quand x est égal à y,
- x != y est vrai quand x est différent de y,
- x > y est vrai quand x est strictement supérieur à y,
- x < y est vrai quand x est strictement inférieur à y,
- x >= y est vrai quand x est supérieur ou égal à y,
- x <= y est vrai quand x est inférieur ou égal à y.

- Condition **if / else** :

L'instruction **if / else** (si/sinon en français) permet un meilleur contrôle du déroulement du programme que la simple instruction if, en permettant de grouper plusieurs tests ensemble.

```
if  var > 10:
    #action A
else:
    #action B
```

Exemple :



Résultat dans la fenêtre Python Shell :

```
===== RESTART: C:\Users\Olivier\Docs\Travail\ArdPyLab\Docs\Python\ifelse.py ====
2
>>>
```

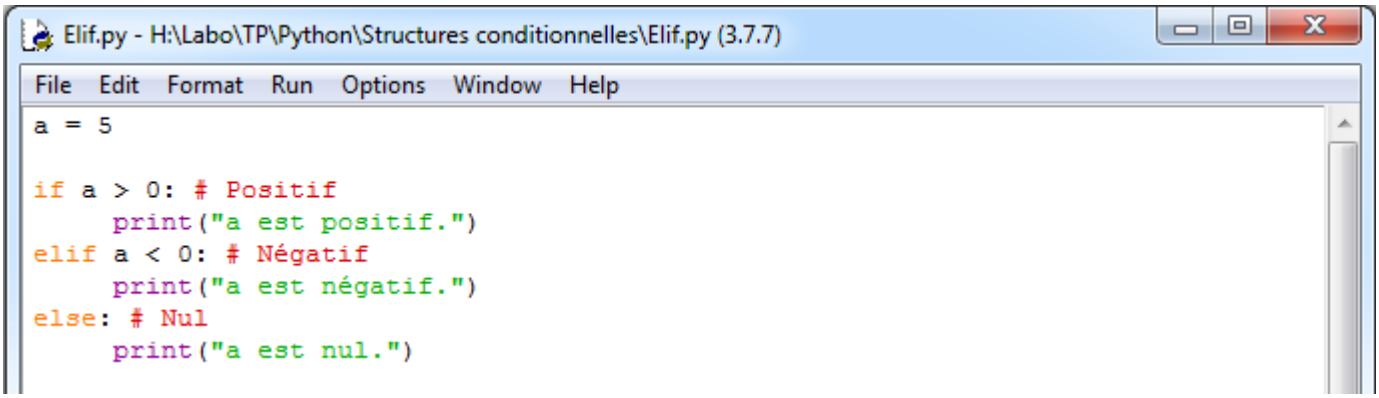
- Condition **elif** :

L'instruction **else** peut contenir un autre test **if**, et donc des tests multiples, mutuellement exclusifs peuvent être réalisés en même temps. On peut alors utiliser le mot clé **elif** à la place de **else : if ...**.

Chaque test sera réalisé après le suivant jusqu'à ce qu'un test VRAI soit rencontré.
Quand une condition vraie est rencontrée, les instructions associées sont réalisées, puis le programme continue son exécution à la ligne suivant l'ensemble de la construction **if/elif**.
Si aucun test n'est VRAI, le bloc d'instructions par défaut **else** est exécuté, s'il est présent, déterminant ainsi le comportement par défaut.

Un bloc **elif** peut être utilisé avec ou sans bloc de conclusion **else**.
Un nombre illimité de branches **elif** est autorisé.

Exemple :



- **AND / OR**

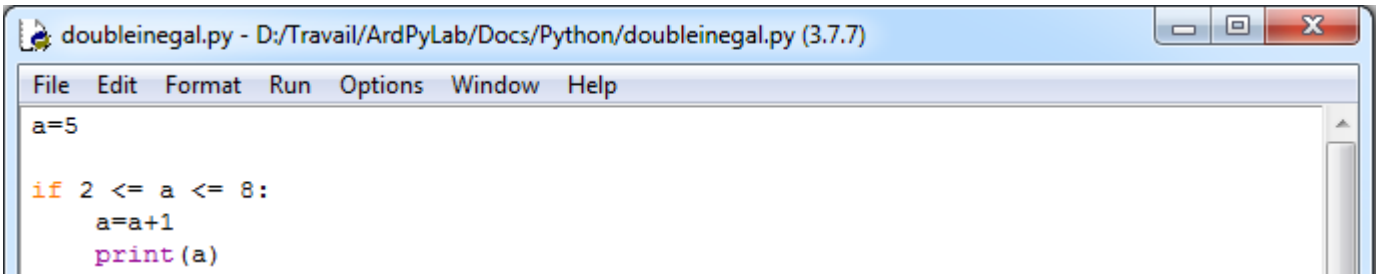
Il est possible d'affiner une condition avec les mots clé **AND** qui signifie " ET " et **OR** qui signifie " OU ".
Ces opérateurs peuvent être utilisés à l'intérieur de la condition d'une instruction if pour associer plusieurs conditions à tester.

```
if var >= 5  and var <= 10 :  # est VRAI seulement si var appartient à l'intervalle [5;10]
      # bloc d'instructions
else:
      # bloc d'instructions
```

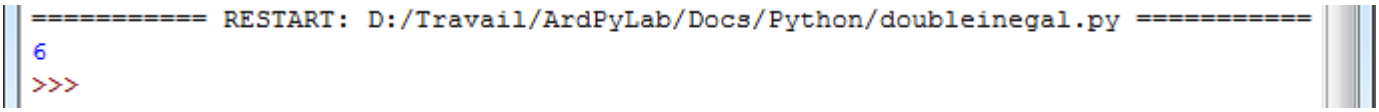
```
if var1 > 0 or var2 > 0 : # est vrai si var1 supérieur à 0 ou si var2 supérieur à 0
    # bloc d'instructions
else:
    # bloc d'instructions
```

Remarque :

Python permet aussi l'enchaînement des comparaisons à l'aide d'une double inégalité.



Résultat dans la fenêtre Python Shell :



- les structures itératives :

. boucles **While**

Les boucles while ("tant que" en anglais) bouclent sans fin, et indéfiniment, jusqu'à ce que la condition ou l'expression testée devienne fausse.
Quelque chose doit modifier la variable testée, sinon la boucle while ne se terminera jamais. C'est généralement une variable incrémentée dans le bloc d'instructions de la boucle.

```
var = 0
while var < 10 :  # tant que la variable est inférieur à 10
    # fait quelque chose 10 fois de suite...
    var += 1 # incrémente la variable
```

Exemple : Affichage d'une table de multiplication

while.py - H:\Labo\TP\Python\Boucles\while.py (3.7.7)

File Edit Format Run Options Window Help

```
nb = 7
i = 0

while i < 10: # Tant que i est strictement inférieure à 10
    print(i + 1, "*", nb, "=", (i + 1) * nb)
    i += 1 # On incrémente i de 1 à chaque tour de boucle
```

Résultat dans la fenêtre Python Shell :

```
================== RESTART: H:\Labo\TP\Python\Boucles\while.py ==================
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
```

Remarques :

Il existe des mots clés qui permettent d'effectuer une rupture dans la boucle itérative :

. **continue** (continue directement à la prochaine itération de la boucle, la boucle est court-circuitée)

Exemple :

Dans l'exemple suivant, tant que la variable i est inférieure à 20, celle-ci est incrémentée de 1 jusqu'à ce qu'elle soit égale à un multiple de 3. Elle est alors incrémentée de 4 et la boucle normale est reprise à l'aide de l'instruction **continue**.

while2.py - H:\Labo\TP\Python\Boucles\while2.py (3.7.7)

File Edit Format Run Options Window Help

```
i = 1
while i < 20:
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue
    print("La variable i =", i)
    i += 1 # Dans le cas classique on ajoute juste 1 à i
```

Résultat dans la fenêtre Python Shell :

```
================= RESTART: H:\Labo\TP\Python\Boucles\while2.py =================
La variable i = 1
La variable i = 2
On incrémente i de 4. i est maintenant égale à 7
La variable i = 7
La variable i = 8
On incrémente i de 4. i est maintenant égale à 13
La variable i = 13
La variable i = 14
On incrémente i de 4. i est maintenant égale à 19
La variable i = 19
```

. **break** (sort de la boucle en cours)

. **pass** (instruction vide – ne fait rien)

Il est parfois pratique d'utiliser une boucle **while** infinie (dont la condition est toujours vraie), et d'utiliser les ruptures de séquences.

```
while True:
    # bloc d'instructions
    if condition : break
```

Exemple :

Dans cet exemple, on demande à l'utilisateur de saisir la lettre 'Q' pour sortir de la boucle.

Break.py - H:\Labo\TP\Python\Boucles\Break.py (3.7.7)

File Edit Format Run Options Window Help

```
while True: # True est toujours vrai -> boucle infinie
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break
```

. boucles **for**

L'utilisation principale de l'instruction **for** est de parcourir un itérable, c'est-à-dire un conteneur que l'on peut parcourir élément par élément, dans l'ordre ou non, suivant son type :

- Parcours d'une chaine de caractères :

for.py - D:/Travail/ArdPyLab/Docs/Python/for.py (3.7.7)

File Edit Format Run Options Window Help

```
s = "azerty"
for lettre in s:
    print(lettre)
```

Résultat dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/for.py ==========
a
z
e
r
t
y
```

- Parcours d'une liste :

for2.py - D:/Travail/ArdPyLab/Docs/Python/for2.py (3.7.7)

File Edit Format Run Options Window Help

```
liste = ["A",2,3,"azerty"]
for elt in liste:
    print(elt)
```

Résultat dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/for2.py ==========
A
2
3
azerty
```

L'instruction **for** peut également être est utilisée pour répéter l'exécution d'un bloc d'instructions à l'aide de la fonction **range()** déjà vu lors de la présentation des listes :

```
for i in range(10):
    # bloc d'instructions
```

Dans cet exemple, le bloc d'instructions sera exécuté 10 fois.

Remarques:

- Une boucle **for** peut également être arrêtée avec l'instruction **break**

Exemple :

Dans cet exemple, la boucle **for** parcours les éléments d'une liste de nombres. Si le nombre lu est supérieur à 15, la boucle est stoppée.

for3.py - D:/Travail/ArdPyLab/Docs/Python/for3.py (3.7.7)

File Edit Format Run Options Window Help

```
liste = [1,5,10,15,20,25]
for i in liste:
    if i > 15:
        print("On stoppe la boucle")
        break
    print(i)
```

Résultat dans la fenêtre Python Shell :

```
============== RESTART: D:/Travail/ArdPyLab/Docs/Python/for3.py ==============
1
5
10
15
On stoppe la boucle
>>>
```

- Les boucles **while** et **for** peuvent posséder une clause **else** qui ne s'exécute que si la boucle se termine normalement, c'est-à-dire sans interruption avec l'instruction **break** :

while_else.py - D:/Travail/ArdPyLab/Docs/Python/while_else.py (3.7.7)

File Edit Format Run Options Window Help

```
liste = ['A', 'B', 'C', 'D']
element = 'B'

i = 0
while i < len(liste):
    if liste[i] == element:
        # element trouvé
        print(element, " est dans la liste, à l'indice: ",i)
        break
    i += 1
else:
    # element non trouvé
    print(element, " n'est pas dans la liste.")
```

Ln: 13 Col: 47

Dans l'exemple ci-dessus, on parcourt une liste à l'aide d'une boucle **while** pour savoir si la variable **element** est dans la liste. Si la variable est trouvée, la boucle est stoppée avec une instruction break :

Résultat dans la fenêtre Python Shell :

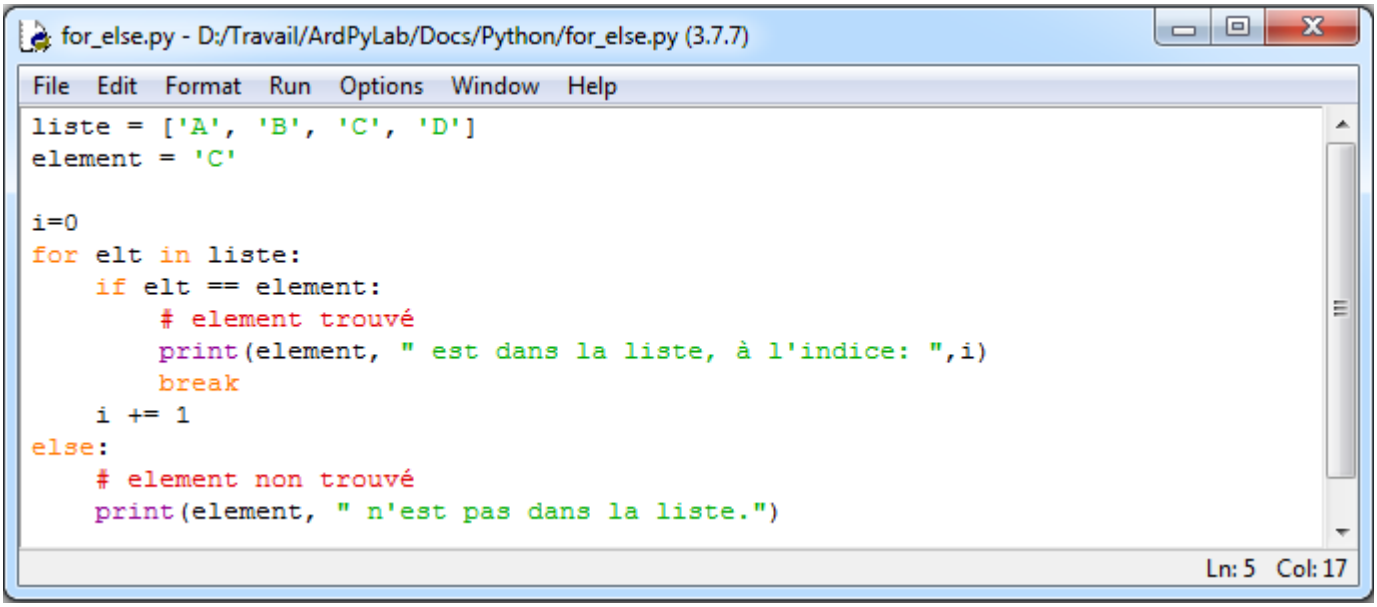
```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/while_else.py ============
B  est dans la liste, à l'indice:  1
>>>
```

, sinon on affiche que la variable n'est pas dans la liste :

Résultat dans la fenêtre Python Shell avec modification de la variable **element = 'E'** :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/while_else.py ==========
E  n'est pas dans la liste.
>>>
```

Il en est de même avec une boucle **for** :



Résultats dans la fenêtre Python Shell :

```
============= RESTART: D:/Travail/ArdPyLab/Docs/Python/for_else.py =============
C  est dans la liste, à l'indice:  2
>>>
```

avec modification de la variable **element = 'E'** :

```
============= RESTART: D:/Travail/ArdPyLab/Docs/Python/for_else.py =============
E  n'est pas dans la liste.
>>>
```

. Les exceptions – Gestion des erreurs dans les scripts

Lorsqu'une instruction d'un script ne se déroule pas correctement (par exemple, une division par zéro), une **exception est levée** ce qui interrompt le contexte d'exécution, pour revenir à un environnement d'exécution supérieur, jusqu'à celui gérant cette exception.

Par défaut, l'environnement supérieur est le shell de commande depuis lequel l'interpréteur Python a été lancé, et le comportement de gestion par défaut est d'afficher l'exception :

```
============= RESTART: D:/Travail/ArdPyLab/Docs/Python/exception.py ============
Traceback (most recent call last):
  File "D:/Travail/ArdPyLab/Docs/Python/exception.py", line 2, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

Pour gérer l'exception, et éviter la fin du programme, il faut utiliser la structure **try** et **except** :

```
try:
        # bloc d'instructions susceptibles d'échouer
except:
        # bloc d'instructions à faire en cas d'échec
```

Toutes les exceptions levées par Python sont des instances de sous-classe de la classe **Exception**.

La hiérarchie des sous-classes offre plusieurs exceptions standard, comme **ValueError** (exception levée quand on tente par exemple de convertir en nombre une chaine de caractères ne représentant pas un nombre) ou **ZeroDivisionError** (quand on tente de diviser un nombre par zéro).

Il est possible de compléter la structure **try except** avec un bloc **else** et un bloc **finally**. Les instructions du bloc **else** ne sont exécutées qu'en l'absence d'erreur et les instructions du bloc **finally** sont toujours effectuées quel que soient les erreurs rencontrées lors de l'exécution du bloc **try** ou en l'absence d'erreur.

La syntaxe complète d'une exception est alors :

```
try:
        ... # séquence normale d'exécution
except exception_1:
```

```
        ... # traitement de l'exception 1
except exception_2:
        ... # traitement de l'exception 2
else:
        ... # bloc d'instructions exécutées en l'absence d'erreur
finally:
        ... # bloc d'instructions toujours exécutées
```

Exemple :

Ce programme demande à l'utilisateur de saisir un nombre et tente de le convertir en flottant et de faire une division avec ce nombre :

exception3.py - D:/Travail/ArdPyLab/Docs/Python/exception3.py (3.7.7)

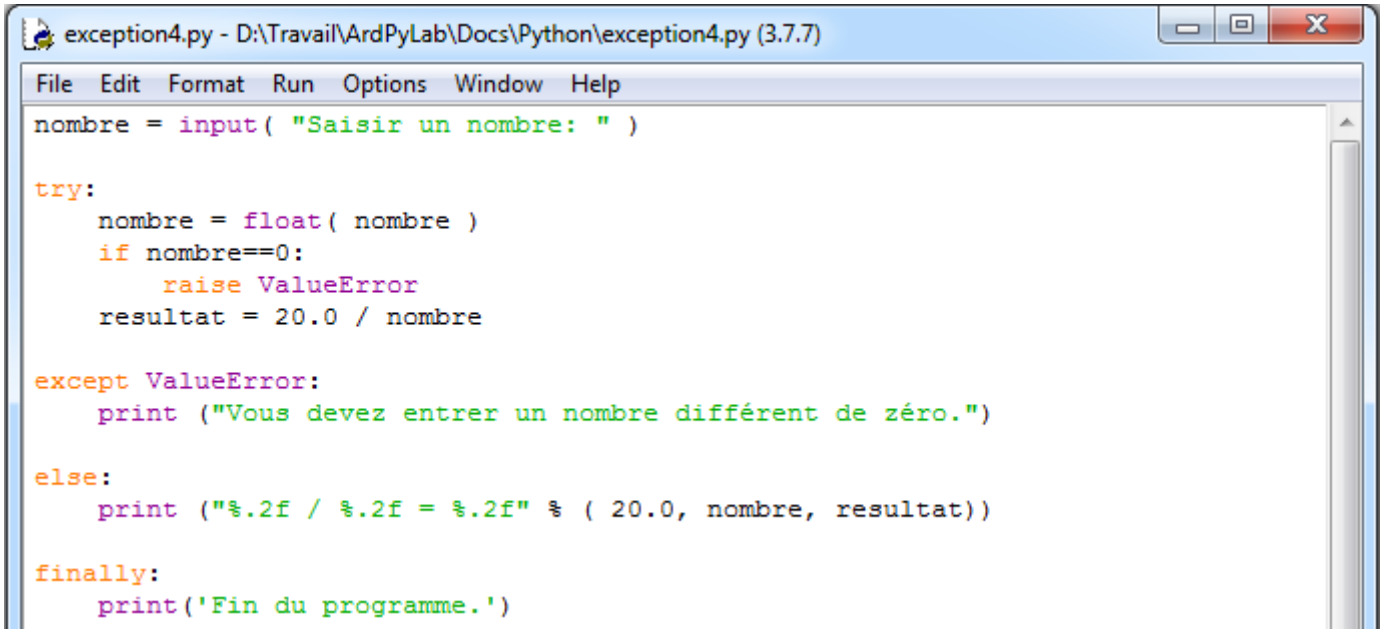
File Edit Format Run Options Window Help

```
nombre = input( "Saisir un nombre: " )

try:
    nombre = float( nombre )
    resultat = 20.0 / nombre

except ValueError:
    print ("Vous devez entrer un nombre.")
except ZeroDivisionError:
    print ("Essai de division par zéro.")

else:
    print ("%.2f / %.2f = %.2f" % ( 20.0, nombre, resultat))

finally:
    print('Fin du programme.')
```

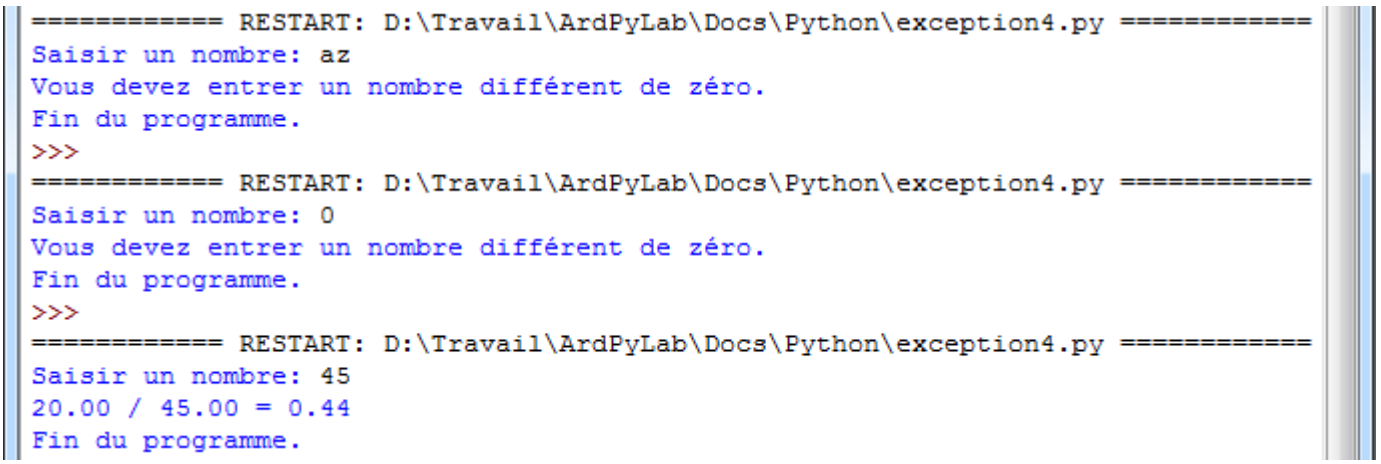
Résultats dans la fenêtre Python Shell :

```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/exception3.py ============
Saisir un nombre: za
Vous devez entrer un nombre.
Fin du programme.
>>>
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/exception3.py ============
Saisir un nombre: 0
Essai de division par zéro.
Fin du programme.
>>>
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/exception3.py ============
Saisir un nombre: 12
20.00 / 12.00 = 1.67
Fin du programme.
```

L'instruction **raise** permet de lever volontairement une exception. Ainsi le script du programme précédent devient :



Après la conversion de la chaine saisie au clavier en nombre, un test est effectué sur le nombre, si celui-ci est égale à 0, l'exception **ValueError** est levée à l'aide de l'instruction **raise**. Et bien-sûr si la chaine ne peut pas être convertie l'exception **ValueError** est également levée (c'est aussi le bloc d'exception du bloc **try**) :



Il est également possible de lever une exception avec l'instruction **assert**. Cette instruction va tester la condition mise juste après, et si elle est fausse, va lever une exception de type **AssertionError,** ce qui donne pour notre exemple :

exception5.py - D:\Travail\ArdPyLab\Docs\Python\exception5.py (3.7.7)

File Edit Format Run Options Window Help

```
nombre = input( "Saisir un nombre: " )

try:
    nombre = float( nombre )
    assert nombre > 0
    resultat = 20.0 / nombre

except AssertionError:
        print("Le nombre saisi doit être supérieur à 0.")

except ValueError:
    print ("Vous devez entrer un nombre.")

else:
    print ("%.2f / %.2f = %.2f" % ( 20.0, nombre, resultat))

finally:
    print('Fin du programme.')
```

Résultats dans la fenêtre Python Shell :

```
========== RESTART: D:\Travail\ArdPyLab\Docs\Python\exception5.py ==========
Saisir un nombre: az
Vous devez entrer un nombre.
Fin du programme.
>>>
========== RESTART: D:\Travail\ArdPyLab\Docs\Python\exception5.py ==========
Saisir un nombre: 0
Le nombre saisi doit être supérieur à 0.
Fin du programme.
>>>
========== RESTART: D:\Travail\ArdPyLab\Docs\Python\exception5.py ==========
Saisir un nombre: 45
20.00 / 45.00 = 0.44
Fin du programme.
```

La structure **Try … Except** est donc très utile pour tout ce qui est vérification des saisies au clavier. Mais contrairement aux scripts précédents, il est préférable que le programme ne s'arrête pas si la saisie au clavier ne satisfait pas au programme.

On utilisera pour cela une boucle **while**, afin de redemander à l'utilisateur une saisie au clavier si la précédente n'est pas adéquate, comme dans l'exemple suivant :

Dans ce programme, on demande à l'utilisateur de saisir un nombre supérieur à 0 et on vérifie si c'est un nombre premier.

exception2.py - D:\Travail\ArdPyLab\Docs\Python\exception2.py (3.7.7)

File Edit Format Run Options Window Help

```
nombresaisi = False

while nombresaisi == False:

    nombre = input("Saisissez un nombre : ")

    try:
        nombresaisi = True
        nombre = int(nombre)
        assert nombre > 0

        i = 2
        while i < nombre and nombre % i != 0:
            i = i + 1

        if i == nombre:
            print("Le nombre", nombre, "est premier.")
        else:
            print("Ce n'est pas un nombre premier.")

        break

    except AssertionError:
        print("Le nombre saisi est inférieur ou égal à 0.")

    except:
        print("vous n'avez pas saisi un nombre!")

    finally:
        nombresaisi = False
```

Résultats dans la fenêtre Python shell :

```
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception2.py ============
Saisissez un nombre : az
vous n'avez pas saisi un nombre!
Saisissez un nombre : 0
Le nombre saisi est inférieur ou égal à 0.
Saisissez un nombre : 15
Ce n'est pas un nombre premier.
>>>
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\exception2.py ============
Saisissez un nombre : 13
Le nombre 13 est premier.
>>>
```

Remarque :

L'instruction **break** du bloc **try** permet, en l'absence d'erreur, de sortir de la boucle **while** et de finir le programme.

. Synthèse structure des scripts Python

Les affectations de variables, les structures conditionnelles et itératives, les gestions d'erreurs sont la base des scripts Python.

Voici un script qui résume tout ce qui a été vu jusqu'à présent. Dans ce programme, l'utilisateur va créer un inventaire de matériel visualisable et modifiable.

Inventaire.py - D:/Travail/ArdPyLab/Docs/Python/Inventaire.py (3.7.7)

File Edit Format Run Options Window Help

```
print("/////////////// INVENTAIRE MATERIEL ///////////////")

Inventaire = {}
Fin =False

print("appuyer sur A pour ajouter un matériel à l'inventaire:")
print("appuyer sur S pour supprimer un matériel de l'inventaire:")
print("appuyer sur V pour afficher la liste de matériel:")
print("appuyer sur Q pour quitter:")
print("////////////////////////////////////////////////")
print(" ")

while Fin == False:

    choix=input()

    if choix.upper() == "A":
        Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
        Quant=input("saisissez la quantité de ce matériel:")
        Inventaire[Matos]=Quant

    if choix.upper() == "V":
        for cle, valeur in Inventaire.items():
            print("{} : {}".format(cle, valeur))

    if choix.upper() == "S":
        element=input("saisissez l'élément à supprimer dans l'inventaire:")
        try:
            del Inventaire[element]
        except:
            print("L'élément que vous voulez supprimer n'existe pas!")

    if choix.upper() == "Q":
        print("Fin du programme")
        Fin = True
```

Résultats dans la fenêtre Python Shell :

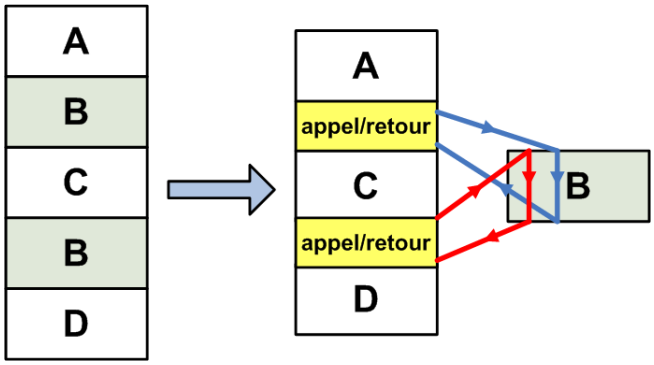
```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/Inventaire.py ==========
/////////////// INVENTAIRE MATERIEL ///////////////
appuyer sur A pour ajouter un matériel à l'inventaire:
appuyer sur S pour supprimer un matériel de l'inventaire:
appuyer sur V pour afficher la liste de matériel:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

A
saisissez le type de matériel à ajouter à l'inventaire:Bécher
saisissez la quantité de ce matériel:20
a
saisissez le type de matériel à ajouter à l'inventaire:Eprouvette
saisissez la quantité de ce matériel:15
v
Bécher : 20
Eprouvette : 15
s
saisissez l'élément à supprimer dans l'inventaire:Bécher
v
Eprouvette : 15
a
saisissez le type de matériel à ajouter à l'inventaire:Erlenmeyer
saisissez la quantité de ce matériel:10
v
Eprouvette : 15
Erlenmeyer : 10
s
saisissez l'élément à supprimer dans l'inventaire:bécher
L'élément que vous voulez supprimer n'existe pas!
q
Fin du programme
```

3.4.2 Les fonctions

Une fonction est un bloc d'instructions que l'on peut appeler à tout endroit d'un programme. Elle est particulièrement utile quand une tâche doit être réalisée plusieurs fois par un programme avec seulement des paramètres différents.

Nous avons déjà vu diverses fonctions prédéfinies : **print()**, **input()**, **range()**, **len()**…

Mais, on peut également créer ses propres fonctions afin d'éviter les répétitions de code et permettre une réutilisation :



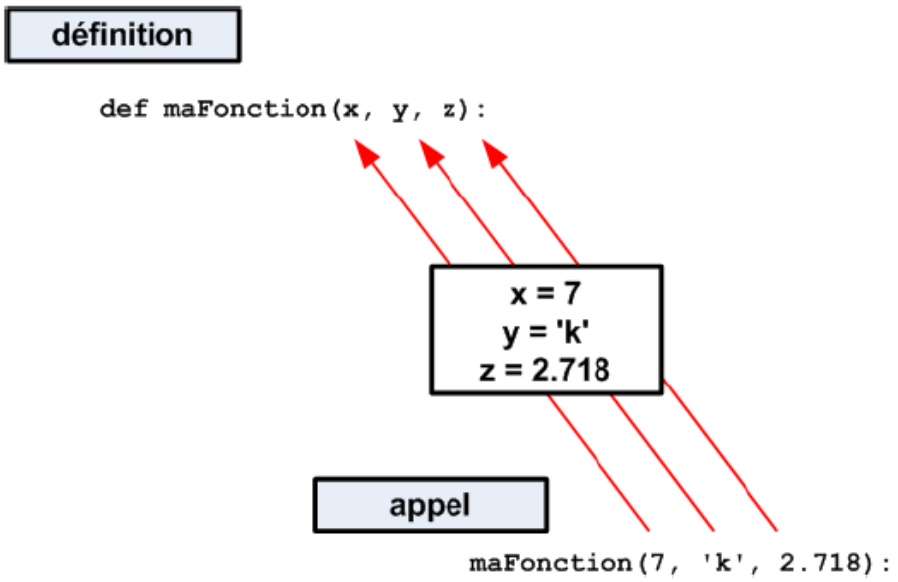
La définition d'une fonction se fait à l'aide du mot clé **def** :

```
def ma_fonction():
    # bloc d'instructions
```

Il est possible de définir des paramètres à la fonction. Ce sont les arguments de la fonction :

def maFonction(x,y,z)

Lors de l'appel de la fonction dans le programme principal, chaque paramètre de l'appel correspond dans l'ordre à chaque argument de la définition de la fonction. La correspondance se fait par affectation :



Exemple :

Dans le programme suivant, La fonction **table** permet d'afficher la table de multiplication d'un nombre. L'argument de la fonction **table** étant ce nombre :

Fonction-table.py - D:/Travail/ArdPyLab/Docs/Python/Fonction-table.py (3.7.7)

File Edit Format Run Options Window Help

```
def table(nb):
    i = 0
    while i < 10:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1

nombresaisi = False

while nombresaisi == False:

    nombre = input("Saisissez un nombre : ")

    try:
        nombresaisi = True
        nombre = int(nombre)
        table(nombre)
        break

    except:
        print("vous n'avez pas saisi un nombre!")

    finally:
        nombresaisi = False
```

Résultats dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/Fonction-table.py ==========
Saisissez un nombre : 12
1 * 12 = 12
2 * 12 = 24
3 * 12 = 36
4 * 12 = 48
5 * 12 = 60
6 * 12 = 72
7 * 12 = 84
8 * 12 = 96
9 * 12 = 108
10 * 12 = 120
```

Les paramètres de la fonction peuvent être nommés et recevoir des valeurs par défaut. Ils peuvent ainsi être donnés dans le désordre et/ou pas en totalité.

Exemple :

Ajoutons à la fonction **table**, l'argument **max**=10 correspondant à la valeur maximale du multiplicateur et donnons une valeur par défaut au nombre à multiplier :

```
def table(nb=1, max=10):
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

L'appel de la fonction pourra se faire ainsi :

. **table(nombre)** : la valeur par défaut de max est utilisée

. **table(nombre1, nombre2)** avec **nombre1** le nombre à multiplier et **nombre2** la valeur maximale du multiplicateur

. **table(nombre1, max=nombre2)**

. **table(nb=nombre1, max=nombre2)**

. **table(max=nombre2, nb=nombre1)**

. table() : les valeurs par défaut de nb et max sont utilisées

. <u>Fonctions avec return</u>

Les fonctions peuvent retourner une ou plusieurs données à l'aide du mot clé **return.** A noter qu'une fonction sans **return**, est plutôt appelée procédure.

<u>Exemple</u> :

De façon à pouvoir utiliser la nouvelle fonction **table**, nous allons modifier le programme d'affichage des tables de multiplication en créant une fonction permettant de saisir un nombre et qui retourne ce nombre :

```
def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre
```

Le seul argument de la fonction est le message à afficher lors de la demande de saisie du nombre à l'aide de la fonction **input()**.

Si l'entrée clavier ne correspond pas à un nombre, l'utilisateur en est informé et la demande de saisi d'un nombre est affichée de nouveau.

Si l'entrée clavier est valide, le nombre saisi est retourné par la fonction.

Le programme d'affichage de la table de multiplication souhaitée est alors :

Fonction-table2.py - D:/Travail/ArdPyLab/Docs/Python/Fonction-table2.py (3.7.7)

File Edit Format Run Options Window Help

```
#Fonction affichage table de multiplication

def table(nb=1, max=10):
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1

# Fonction saisie d'un nombre

def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre

# Appels des fonctions

nombre=saisie("Veuillez saisir le nombre à multiplier:")
multimax=saisie("Veuillez saisir la valeur maximale du multiplicateur:")

table(nombre,multimax)
```

Résultats dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/Fonction-table2.py =========
Veuillez saisir le nombre à multiplier:az
vous n'avez pas saisi un nombre!
Veuillez saisir le nombre à multiplier:5
Veuillez saisir la valeur maximale du multiplicateur:d
vous n'avez pas saisi un nombre!
Veuillez saisir la valeur maximale du multiplicateur:9
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
```

Remarque :

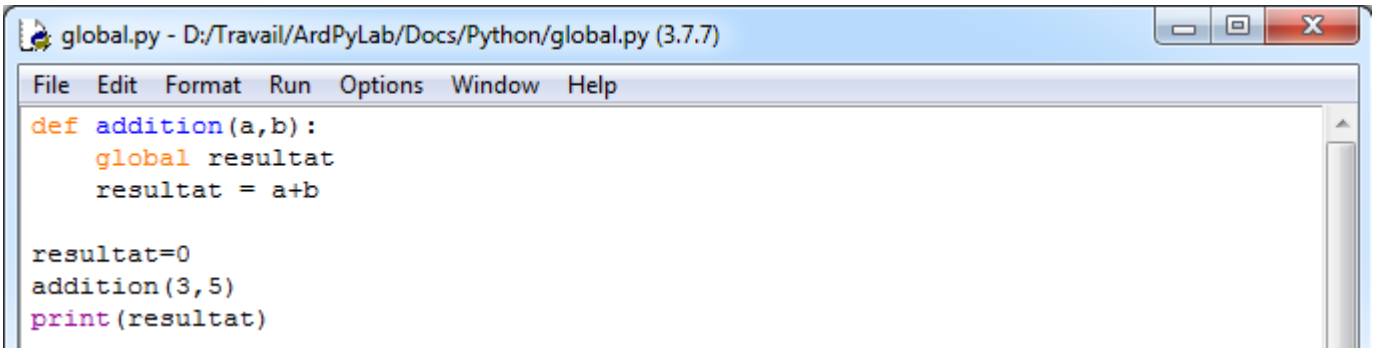
Les variables à l'intérieur du corps d'une fonction ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales**.

Une variable locale peut avoir le même nom qu'une variable du programme principal mais elle reste néanmoins indépendante.

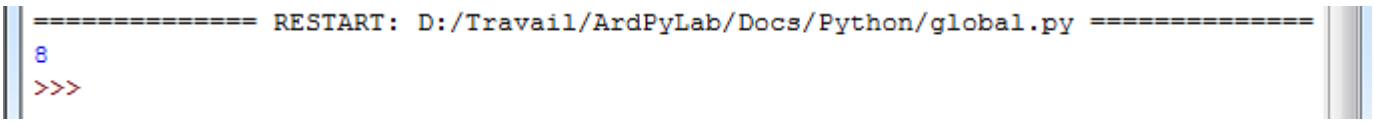
Le contenu des variables locales est inaccessible depuis l'extérieur de la fonction.

Les variables définies à l'extérieur d'une fonction sont des **variables globales**. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

Pour modifier une variable globale au sein d'une fonction, il faut utiliser l'instruction **global**. Cette instruction permet d'indiquer quelles sont les variables à traiter globalement :



Résultats dans la fenêtre Python Shell :



. Fonctions lambda

Pour des fonctions très courtes, on peut utiliser des fonctions anonymes, connues aussi sous le nom de fonctions lambda.

Prenons l'exemple d'une fonction retournant la valeur de l'addition de deux nombres :

```
def addition(a,b):
    return a+b
```

On peut écrire cette fonction en une seule ligne, avec le mot clé **lambda** :

```
Addition = lambda x,y: x+y
```

A noter cependant qu'avec les fonctions lambda :

. On ne peut les écrire que sur une seule ligne.

. On ne peut pas avoir plus d'une instruction dans la fonction.

Voici le programme permettant d'afficher le résultat de l'addition :

Lambda.py - D:/Travail/ArdPyLab/Docs/Python/Lambda.py (3.7.7)

File Edit Format Run Options Window Help

```
def addition(a,b):
    return a+b

Addition = lambda x,y: x+y

def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre

Nb1 = saisie("Saisissez un premier nombre : ")
Nb2 = saisie("Saisissez un deuxième nombre : ")

print ("la somme des deux nombres est :",addition(Nb1,Nb2))
print ("Fonction lambda - la somme des deux nombres est :",Addition(Nb1,Nb2))
```

Résultats dans la fenêtre Python Shell :

```
============== RESTART: D:/Travail/ArdPyLab/Docs/Python/Lambda.py ==============
Saisissez un premier nombre : 45
Saisissez un deuxième nombre : 5
la somme des deux nombres est : 50
Fonction lambda - la somme des deux nombres est : 50
>>>
```

3.4.3 Les fichiers

Pour sauvegarder des données, il peut être intéressant de les stocker dans des fichiers qu'il sera possible de lire ultérieurement et éventuellement modifier.

En premier, il faut ouvrir ou créer un fichier avec la fonction **open()**. Cette fonction prend en premier paramètre le chemin du fichier et en second paramètre le type d'ouverture :

fichier = open(”chemin”, ”type d'ouverture”)

On peut également préciser l'encodage du fichier :

fichier = open(”chemin”, ”type d'ouverture”, encoding="utf-8")

(UTF-8 est un encodage universel qui réunit les caractères utilisés par toutes les langues)

Les types d'ouvertures sont :

. **”r”**, pour une ouverture en lecture (READ).

. **”w”**, pour une ouverture en écriture (WRITE), à chaque ouverture le contenu du fichier est écrasé. Si le fichier n'existe pas python le crée.

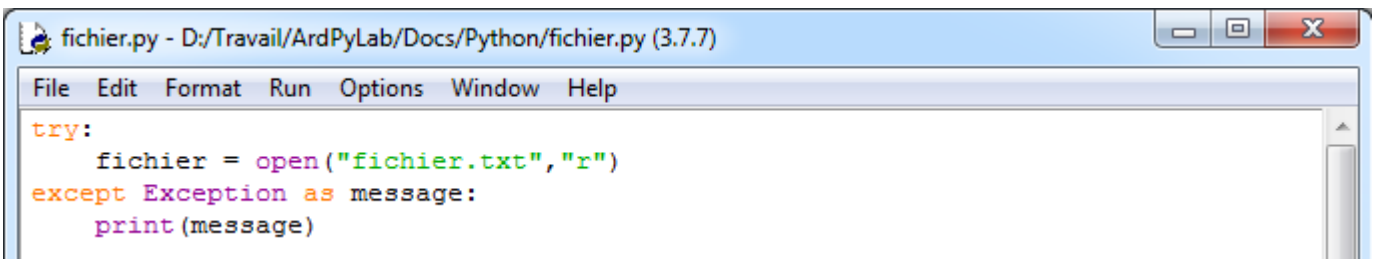
. **”a”**, pour une ouverture en mode ajout à la fin du fichier (APPEND). Si le fichier n'existe pas python le crée.

. **”b”**, pour une ouverture en mode binaire.

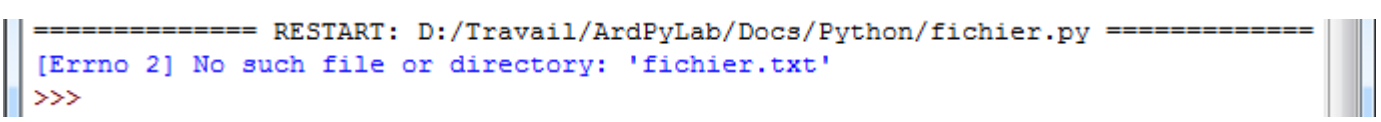
. **”t”**, pour une ouverture en mode texte.

. **”x”**, crée un nouveau fichier et l'ouvre pour écriture.

Pour vérifier que l'ouverture du fichier se fait correctement, il faut traiter une exception de type Exception (elle peut fournir une description de l'erreur) :



Résultat dans la fenêtre Python Shell :

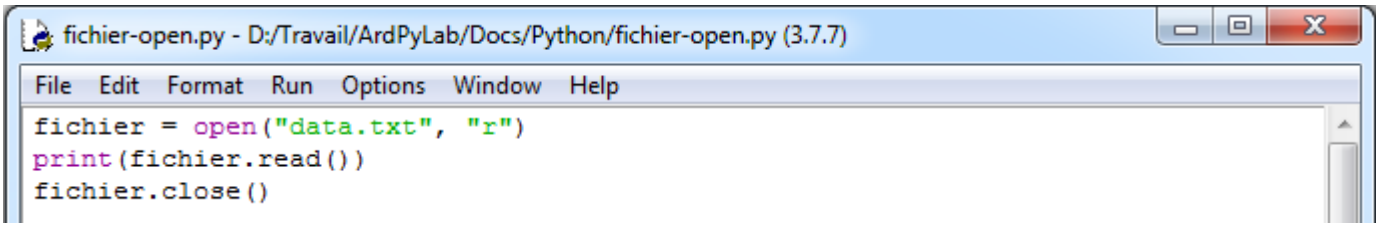


Après ouverture du fichier et une fois les instructions sur le fichier terminées, il faut le fermer. Pour cela, on utilise la méthode **close() :**

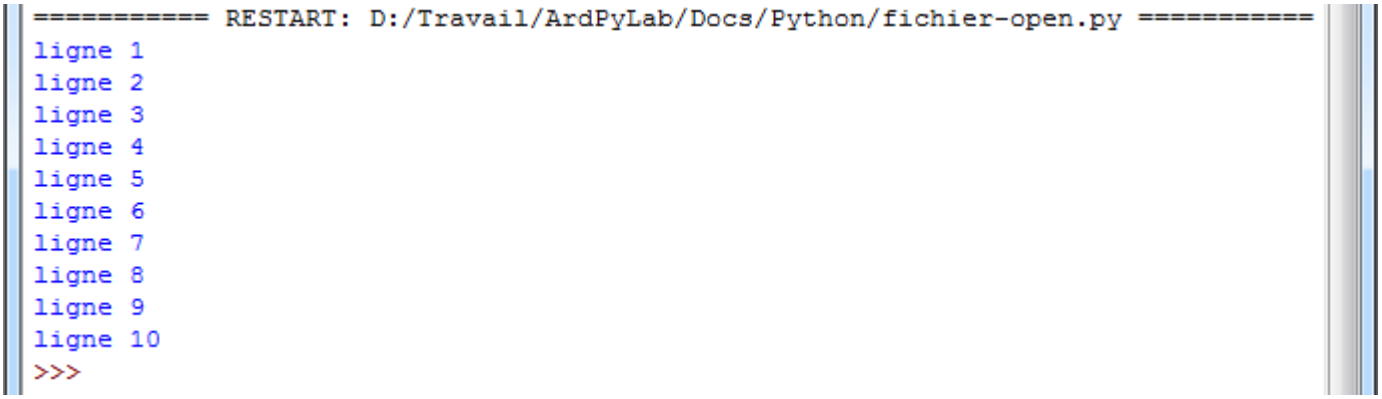
fichier.close()

. Lecture d'un fichier

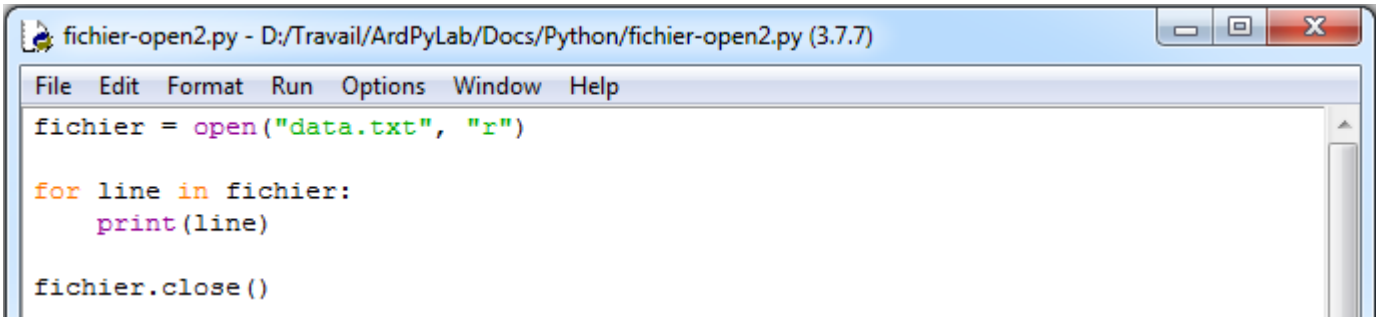
- Pour afficher tout le contenu d'un fichier, on peut utiliser la méthode **read** sur l'objet **fichier** :



Résultat dans la fenêtre Python Shell :



L'objet de type **file** est un objet qui peut se comporter comme une liste, ce qui permet d'utiliser également une boucle **for** :



Attention, lorsque l'on récupère une ligne d'un fichier, c'est une chaîne de caractères qui se termine par le caractère '**\n**' de fin de ligne.

Ou bien simplement, la méthode **readline()** :

fichier-open3.py - D:/Travail/ArdPyLab/Docs/Python/fichier-open3.py (3.7.7)

File Edit Format Run Options Window Help

```
fichier = open("data.txt", "r")

while True:
    line = fichier.readline() #renvoie une ligne avec le \n à la fin
    if not line: break
    print(line)


fichier.close()
```

Ces deux façons de lire un fichier donnent cet affichage dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/fichier-open3.py ==========
ligne 1

ligne 2

ligne 3

ligne 4

ligne 5

ligne 6

ligne 7

ligne 8

ligne 9

ligne 10
>>>
```

. Ecrire dans un fichier

Pour écrire dans un fichier, il faut au préalable l'ouvrir ou le créer en mode **"w"**, **"a"** ou **"x"**.

fichier-write.py - D:/Travail/ArdPyLab/Docs/Python/fichier-write.py (3.7.7)

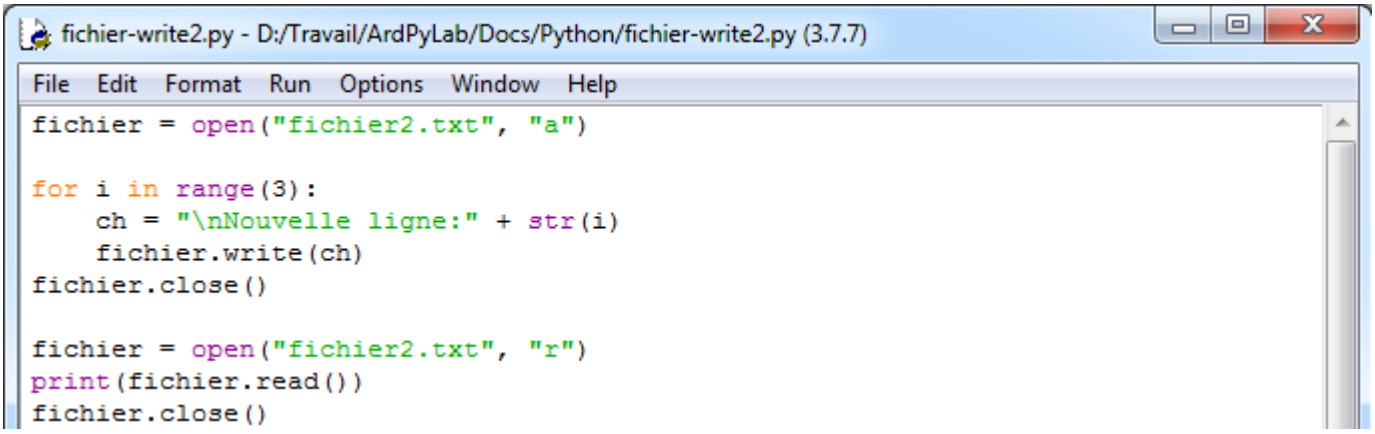
File Edit Format Run Options Window Help

```
fichier = open("fichier2.txt", "a")
fichier.write("Nouvelle ligne")
fichier.close()

fichier = open("fichier2.txt", "r")
print(fichier.read())
fichier.close()
```

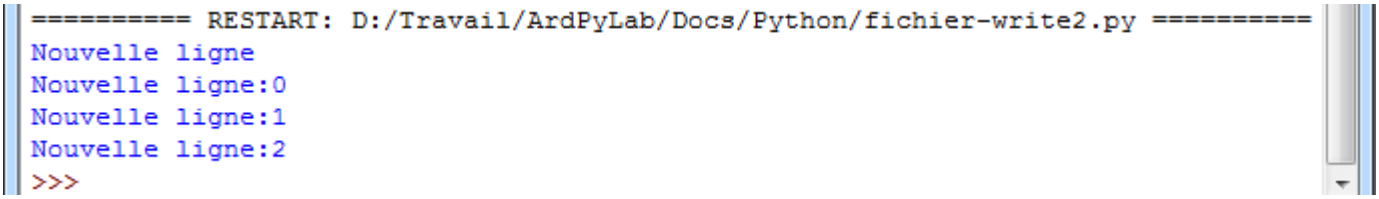
Résultat dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/fichier-write.py ==========
Nouvelle ligne
```

A noter que pour le monde d'ouverture **”a”**, pour écrire à la ligne, on peut utiliser le saut de ligne \n :

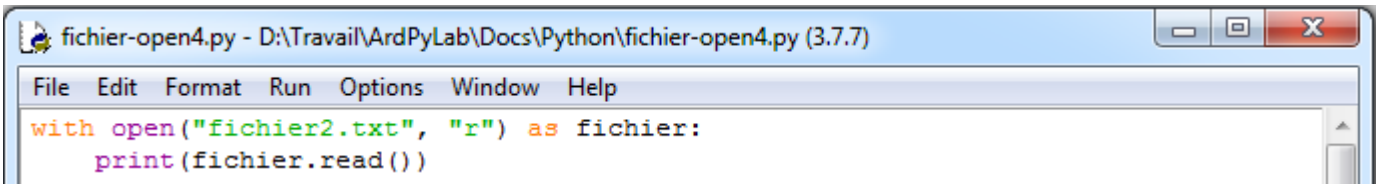


Résultat dans la fenêtre Python Shell :

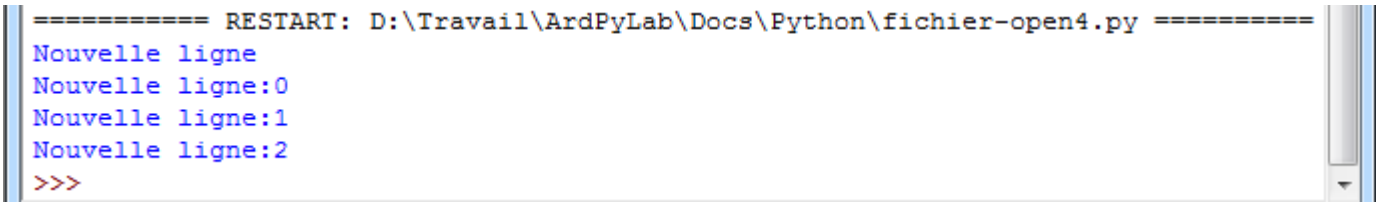


. Autre méthode d’ouverture de fichiers

Une ouverture de fichiers avec le mot clé **with** est également possible et cette méthode présente l’avantage de ne pas être obligé de fermer le fichier après traitement par le programme.



Résultat dans la fenêtre Python Shell :



Remarque :

Si l’on ne précise pas l’emplacement où l’on veut créer un fichier, celui-ci sera créé dans le répertoire courant (en général, dossier du script python).

. Synthèse sur la manipulation des fichiers

Le script ci-dessous résume les principales manipulations qu'il est possible d'effectuer avec un fichier :

lesfichiers.py - D:/Travail/ArdPyLab/Docs/Python/lesfichiers.py (3.7.7)

File Edit Format Run Options Window Help

```
def ManipFichier(chemin, mode, text=''):
    try:
        with open(chemin, mode) as fichier:
            FichierOuvert = True
            if mode=='r':
                texte = fichier.read()
                print(texte)
            if mode=='w':
                fichier.write(text)
            if mode=='a':
                texte='\n'+text
                fichier.write(texte)

    except Exception as message:
        print(message)

Fin = False

print("//////////////////////LES FICHIERS//////////////////////////")
print("Appuyer sur R pour ouvrir et lire le fichier:")
print("Appuyer sur E pour ouvrir et écrire dans un nouveau fichier:")
print("appuyer sur A pour ouvrir et ajouter du texte au fichier:")
print("appuyer sur Q pour quitter:")
print("////////////////////////////////////////////////////////////")
print(" ")

while Fin == False:

    choix=input()

    if choix.upper() == "R": ManipFichier('lesfichiers.txt', 'r')

    if choix.upper() == "E":
        texte=input("Saisissez le texte à sauvegarder:")
        ManipFichier('lesfichiers.txt', 'w', texte)

    if choix.upper() == "A":
        texte=input("Saisissez le texte à sauvegarder:")
        ManipFichier('lesfichiers.txt', 'a', texte)

    if choix.upper() == "Q":
        print("Fin du programme")
        Fin = True
```

Ln: 47 Col: 17

Les taches effectuées par la fonction **ManipFichier()** dépendent des arguments de la fonction. En effet, après avoir vérifié que l'ouverture du fichier, dont le chemin est en

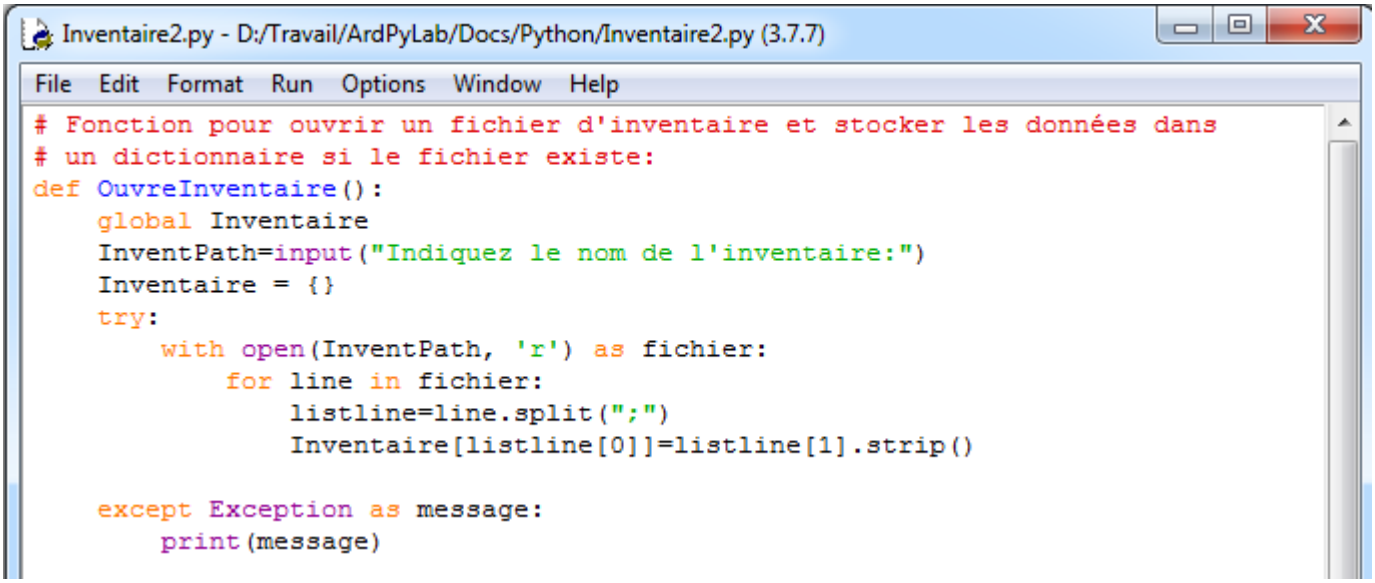
argument, se fait correctement (structure **try except**), le fichier est soit lu, soit créé et mis à jour, soit ouvert et mis à jour, en fonction des valeurs des arguments **mode** et **text**.

Résultats dans la fenêtre Python Shell :

```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/lesfichiers.py ===========
//////////////////////////LES FICHIERS//////////////////////////
Appuyer sur R pour ouvrir et lire le fichier:
Appuyer sur E pour ouvrir et écrire dans un nouveau fichier:
appuyer sur A pour ouvrir et ajouter du texte au fichier:
appuyer sur Q pour quitter:
////////////////////////////////////////////////////////////////

r
[Errno 2] No such file or directory: 'lesfichiers.txt'
e
Saisissez le texte à sauvegarder:ligne1
r
ligne1
a
Saisissez le texte à sauvegarder:ligne2
r
ligne1
ligne2
e
Saisissez le texte à sauvegarder:new
r
new
q
Fin du programme
```

. Exemple d'application

Maintenant que nous savons sauvegarder des données et définir des fonctions, nous allons pouvoir modifier le programme de création d'inventaire de façon à pouvoir enregistrer les modifications qui lui sont apportées à l'aide de fonctions définies pour chaque action sur l'inventaire.



```
# Fonction pour ajouter une ligne à l'inventaire ouvert ou initialement vide:
def AjoutMatos():
    global Inventaire
    Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
    Quant=input("saisissez la quantité de ce matériel:")
    Inventaire[Matos]=Quant

#Fonction pour effacer une ligne de l'inventaire en cours:
def EffaceMatos():
    global Inventaire
    element=input("saisissez l'élément à supprimer dans l'inventaire:")
    try:
        del Inventaire[element]
    except:
        print("L'élément que vous voulez supprimer n'existe pas!")

#Fonction pour afficher l'inventaire en cours:
def ReadInventaire():
    global Inventaire
    for cle, valeur in Inventaire.items():
        print("{} : {}".format(cle, valeur))

#Fonction pour sauvegarder l'inventaire en cours:
def SaveInventaire():
    global Inventaire
    InventPath=input("Indiquez le nom de sauvegarde de l'inventaire:")
    with open(InventPath, 'w') as fichier:
        for cle, valeur in Inventaire.items():
            fichier.write("{};{}".format(cle, valeur))
            fichier.write("\n")

# initialisation des variables:
Inventaire = {}
Fin =False

print("/////////////// INVENTAIRE MATERIEL ///////////////")
print("appuyer sur O pour ouvrir un inventaire:")
print("appuyer sur A pour ajouter un matériel à l'inventaire:")
print("appuyer sur S pour supprimer un matériel de l'inventaire:")
print("appuyer sur V pour afficher la liste de matériel:")
print("appuyer sur E pour enregistrer l'inventaire:")
print("appuyer sur Q pour quitter:")
print("////////////////////////////////////////////////")
print(" ")

# programme principal:
while Fin == False:

    # attente d'un choix:
    choix=input()

    # Action en fonction de l'entrée clavier:
    if choix.upper() == "O": OuvreInventaire()
    if choix.upper() == "A": AjoutMatos()
    if choix.upper() == "V": ReadInventaire()
    if choix.upper() == "S": EffaceMatos()
    if choix.upper() == "E": SaveInventaire()
    if choix.upper() == "Q":
        print("Fin du programme")
        Fin = True
```

Résultats dans la fenêtre Python Shell :

```
== RESTART: C:\Users\Olivier\Docs\Travail\ArdPyLab\Docs\Python\Inventaire2.py ==
////////////// INVENTAIRE MATERIEL //////////////
appuyer sur O pour ouvrir un inventaire:
appuyer sur A pour ajouter un matériel à l'inventaire:
appuyer sur S pour supprimer un matériel de l'inventaire:
appuyer sur V pour afficher la liste de matériel:
appuyer sur E pour enregistrer l'inventaire:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

o
Indiquez le nom de l'inventaire:invent
v
bécher : 10
eprouvette : 15
o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
eprouvette : 5
s
saisissez l'élément à supprimer dans l'inventaire:eprouvette
v
bécher : 10
erlenmeyer : 20
e
Indiquez le nom de sauvegarde de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
o
Indiquez le nom de l'inventaire:invent
v
bécher : 10
eprouvette : 15
o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
q
Fin du programme
>>>
```

```
== RESTART: C:\Users\Olivier\Docs\Travail\ArdPyLab\Docs\Python\Inventaire2.py ==
////////////// INVENTAIRE MATERIEL //////////////
appuyer sur O pour ouvrir un inventaire:
appuyer sur A pour ajouter un matériel à l'inventaire:
appuyer sur S pour supprimer un matériel de l'inventaire:
appuyer sur V pour afficher la liste de matériel:
appuyer sur E pour enregistrer l'inventaire:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

o
Indiquez le nom de l'inventaire:invent2
a
saisissez le type de matériel à ajouter à l'inventaire:Pipette
saisissez la quantité de ce matériel:25
v
bécher : 10
erlenmeyer : 20
Pipette : 25
q
Fin du programme
```

3.4.4 Les modules – Les packages

Nous avons vu que dans un script Python (fichier avec une extension **.py**), il était possible de définir des fonctions réutilisables afin d'éviter les répétitions de code.
Jusqu'à présent, dans les scripts que nous avons étudiés, les fonctions étaient toujours définies au début du script.
Il est cependant possible d'effectuer la définition des fonctions dans un fichier séparé pour ensuite utiliser les fonctions dans un script principal.

. Les modules

Un programme Python est donc généralement composé de plusieurs fichiers sources avec une extension .**py**, appelés **modules** indépendants les uns des autres et pouvant être réutilisés dans d'autres programmes.

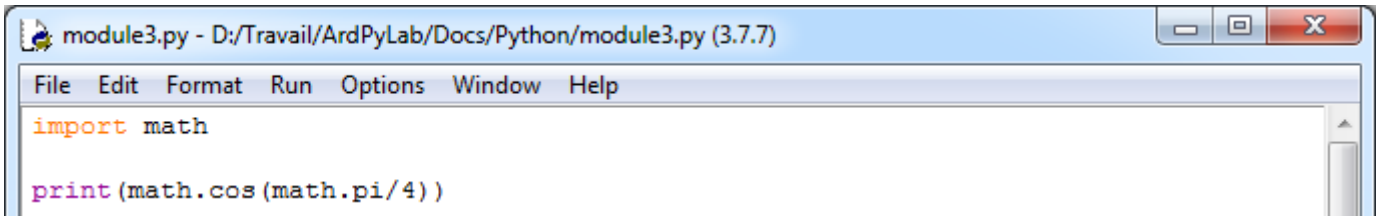
Un module rassemble les fonctions utilisées par le programme principal dans un même fichier situé dans le répertoire d'exécution du programme (pour les modules personnels).

Pour pouvoir utiliser les fonctions du module, le programme principal doit les importer.

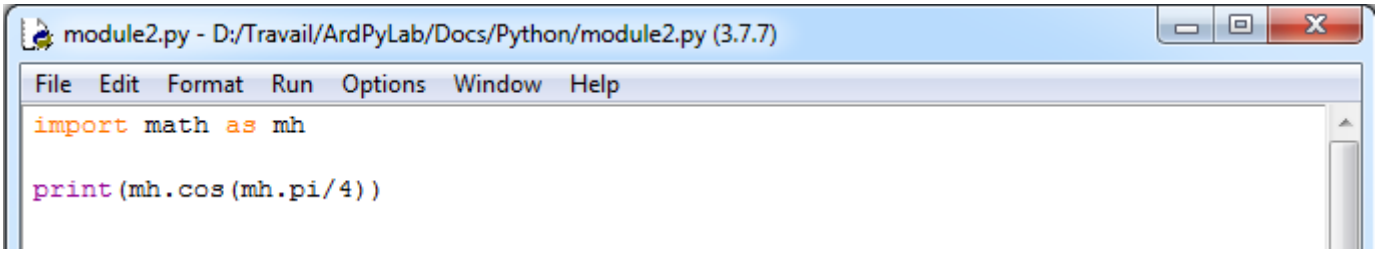
Deux syntaxes sont possibles pour l'importation d'un module :

. la commande **import nom_module** importe la totalité des objets du module.

Exemple : import math



Le module peut être importé sous un autre nom, un alias. On utilise alors cet alias pour appeler les fonctions :



. la commande **from nom_module import obj1, obj2**... n'importe que les objets **obj1**, **obj2**... du module :

Exemple: from math import pi, sin, log

Dans cet exemple, seules les fonctions **cos** et **pi** du module math sont importées :

module4.py - D:/Travail/ArdPyLab/Docs/Python/module4.py (3.7.7)

File Edit Format Run Options Window Help

```
from math import cos, pi

print(cos(pi/4))
```

Ou toutes les fonctions du module :

module.py - D:/Travail/ArdPyLab/Docs/Python/module.py (3.7.7)

File Edit Format Run Options Window Help

```
from math import *

print(cos(pi/4))
```

(Le caractère **'*'** permet d'importer toutes les fonctions du module.)

Exemple :

Reprenons le programme d'affichage de la table de multiplication en créant un module appelé **table.py** contenant les fonctions du programme.

table.py - D:/Travail/ArdPyLab/Docs/Python/table.py (3.7.7)

File Edit Format Run Options Window Help

```
#Fonction affichage table de multiplication

def table(nb=1, max=10):
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1

# Fonction saisie d'un nombre

def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre
```

Ln: 20 Col: 17

Le programme principal importe toutes les fonctions du module table et les utilise :

table_main.py - D:/Travail/ArdPyLab/Docs/Python/table_main.py (3.7.7)

File Edit Format Run Options Window Help

```
from table import *

nombre=saisie("Veuillez saisir le nombre à multiplier:")
multimax=saisie("Veuillez saisir la valeur maximale du multiplicateur:")

table(nombre,multimax)
```

Ln: 6 Col: 22

Résultat dans la fenêtre Python Shell :

```
============ RESTART: D:/Travail/ArdPyLab/Docs/Python/table_main.py ============
Veuillez saisir le nombre à multiplier:5
Veuillez saisir la valeur maximale du multiplicateur:9
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
```

. Les packages

Quand on a un grand nombre de modules, il est intéressant de les organiser dans des dossiers. Un dossier qui rassemble des modules est appelé un **package** (paquets en français). Ce dossier doit contenir un fichier nommé **__init__.py** vide ou décrivant l'arborescence du package. Le fichier **__init__.py** même vide est essentiel pour que Python considère les dossiers le contenant comme des paquets.

Exemple :

Toujours avec programme d'affichage de la table de multiplication, nous allons créer un module **table.py** contenant la fonction d'affichage de la table :

table.py - D:\Travail\ArdPyLab\Docs\Python\fonctions\table.py (3.7.7)

File Edit Format Run Options Window Help

```
#Fonction affichage table de multiplication

def table(nb=1, max=10):
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Et un module **saisie.py** contenant la fonction de saisie d'un nombre :

saisie.py - D:\Travail\ArdPyLab\Docs\Python\fonctions\saisie.py (3.7.7)

File Edit Format Run Options Window Help

```
# Fonction saisie d'un nombre

def saisie(message):
    nombresaisi = False
    while nombresaisi == False:
        nombre = input(message)
        try:
            nombre = int(nombre)
            nombresaisi = True
        except:
            print("vous n'avez pas saisi un nombre!")
    return nombre
```

Ces deux modules ainsi que le fichier **__init__.py** sont situés dans un dossier nommé **fonctions**. Dans ce cas simple, le fichier **__init__.py** peut être vide, mais il peut également décrire l'arborescence du dossier **fonctions** :

init.py - D:\Travail\ArdPyLab\Docs\Python\fonctions_init_.py (3.7.7)

File Edit Format Run Options Window Help

```
fonctions/
    _init_.py
    saisie.py
    table.py
```

Le dossier **fonctions** est situé dans le répertoire d'exécution du programme principal qui va importer la fonction **table** du module **table.py** et la fonction **saisie** du module **saisie.py** :

table_main2.py - D:/Travail/ArdPyLab/Docs/Python/table_main2.py (3.7.7)

File Edit Format Run Options Window Help

```
# importation des modules du package

from fonctions.table import table  # ou from fonctions.table import *
from fonctions.saisie import saisie # ou from fonctions.table import *

# appel des fonctions

nombre=saisie("Veuillez saisir le nombre à multiplier:")
multimax=saisie("Veuillez saisir la valeur maximale du multiplicateur:")

table(nombre,multimax)
```

Résultat dans la fenêtre Python Shell :

```
========== RESTART: D:/Travail/ArdPyLab/Docs/Python/table_main2.py ==========
Veuillez saisir le nombre à multiplier:5
Veuillez saisir la valeur maximale du multiplicateur:5
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
```

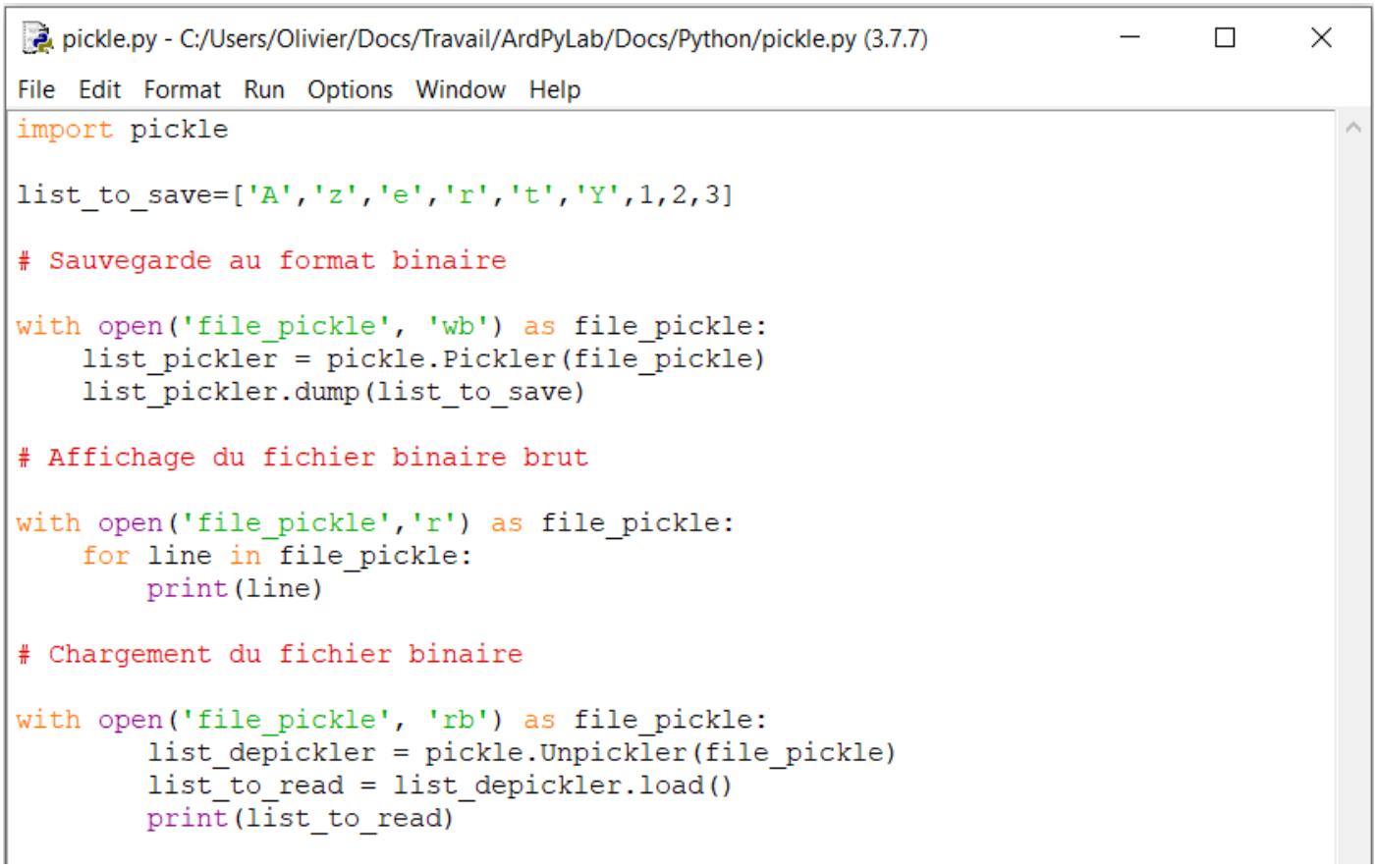
. La bibliothèque standard de Python

Par défaut, Python dispose de nombreux packages et modules intégrés à sa bibliothèque standard. On peut citer :

. **random** : fonctions permettant de travailler avec des valeurs aléatoires
. **math** : toutes les fonctions utiles pour les opérations mathématiques (cosinus,sinus,exp,etc.)
. **sys** : fonctions systèmes
. **os** : fonctions permettant d'interagir avec le système d'exploitation
. **time** : fonctions permettant de travailler avec le temps
. **tkinter** : interface graphique
…

Par exemple, le module **pickle** permet de sauvegarder dans un fichier au format binaire, n'importe quel objet Python (une liste, un dictionnaire, un tuple, etc…). C'est une alternative intéressante à la sauvegarde des données dans un fichier texte.

Le script ci-dessous enregistre une liste appelée **list_to_save** dans un fichier binaire nommé **file_pickle**, puis ouvre le fichier en mode lecture normal et l'affiche dans la fenêtre Python Shell pour montrer qu'il s'agit bien d'un fichier binaire, et enfin charge le fichier en mode lecture binaire et l'affiche dans la fenêtre Python Shell :



Déroulement du programme

- Sauvegarde au format binaire :

Le fichier **file_pickle** est d'abord ouvert ou créé en mode écriture binaire **('wb')**. Avec la méthode **Pickler** du module **pickle,** on crée l'objet **list_pickler** rattaché au fichier binaire. La méthode **dump** appliquée à cet objet permet d'enregistrer au format binaire la liste **list_to_save** dans le fichier **file_pickle**.

- Affichage du fichier binaire brut :

Le fichier est ouvert en mode lecture normal **('r')** et affiché dans la fenêtre Python Shell.

- Chargement du fichier binaire :

Le fichier **file_pickle** est d'abord ouvert en mode lecture binaire **('rb')**. Avec la méthode **Unpickler** du module **pickle,** on crée l'objet **list_depickler** rattaché au fichier binaire ouvert. La méthode **load** appliquée à cet objet permet d'affecter la liste sauvegardée en binaire à la liste **list_to_read** qui est ensuite affichée.

Résultats dans la fenêtre Python Shell

```
===== RESTART: C:/Users/Olivier/Docs/Travail/ArdPyLab/Docs/Python/pickle.py ====
€□]q (X□  Aq□X□  zq X□  eq□X□  rq□X□  tq□X□  Yq□K□K K□e.
['A', 'z', 'e', 'r', 't', 'Y', 1, 2, 3]
>>>
```

Remarque :

Pour la méthode **dump**, le fichier doit toujours être ouvert en mode '**wb**' afin d'écraser le contenu précédent si le fichier existe déjà.

Exemple d'application :

Nous allons appliquer cette méthode de sauvegarde à notre programme de gestion d'inventaire.

Dans le script suivant, des fonctions d'ouverture et de sauvegarde au format binaire d'un dictionnaire nommé **inventaire** sont définies et utilisées pour afficher l'inventaire et enregistrer les modifications qui lui sont apportées (ajout ou suppression de lignes).

Invent-pickle.py - C:/Users/Olivier/Docs/Travail/ArdPyLab/Docs/Python/Invent-pickle.py (3.7.7)

File Edit Format Run Options Window Help

```
import pickle

# Fonction pour ouvrir le fichier d'inventaire et stocker les données dans
# un dictionnaire si le fichier existe:
def OuvreInventaire():
    global Inventaire
    Inventaire = {}
    try:
        with open('fileInvent', 'rb') as fichierInvent:
            Invent_depickler = pickle.Unpickler(fichierInvent)
            Inventaire = Invent_depickler.load()
    except Exception as message:
        print(message)

#Fonction pour sauvegarder l'inventaire en cours:
def SaveInventaire():
    global Inventaire
    with open('fileInvent', 'wb') as fichierInvent:
        Invent_pickler = pickle.Pickler(fichierInvent)
        Invent_pickler.dump(Inventaire)

# Fonction pour ajouter une ligne à l'inventaire ouvert ou initialement vide
# et enregistrer les modifications:
def AjoutMatos():
    global Inventaire
    Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
    Quant=input("saisissez la quantité de ce matériel:")
    Inventaire[Matos]=Quant
    SaveInventaire()

#Fonction pour effacer une ligne de l'inventaire en cours
# et enregistrer les modifications:
def EffaceMatos():
    global Inventaire
    element=input("saisissez l'élément à supprimer dans l'inventaire:")
    try:
        del Inventaire[element]
        SaveInventaire()
    except:
        print("L'élément que vous voulez supprimer n'existe pas!")

#Fonction pour afficher l'inventaire en cours:
def ReadInventaire():
    global Inventaire
    OuvreInventaire()
    for cle, valeur in Inventaire.items():
        print("{} : {}".format(cle, valeur))


# initialisation des variables:
Inventaire = {}
Fin =False

print("////////////////// INVENTAIRE MATERIEL /////////////////")
print("appuyer sur A pour ajouter un matériel à l'inventaire.")
print("appuyer sur S pour supprimer un matériel de l'inventaire.")
print("appuyer sur V pour afficher la liste de matériel.")
print("appuyer sur Q pour quitter.")
print("////////////////////////////////////////////////////////")
print(" ")

# programme principal:
while Fin == False:

    # attente d'un choix:
    choix=input()

    # Action en fonction de l'entrée clavier:
    if choix.upper() == "A": AjoutMatos()
    if choix.upper() == "V": ReadInventaire()
    if choix.upper() == "S": EffaceMatos()
    if choix.upper() == "Q":
        print("Fin du programme")
        Fin = True
```

Résultats dans la fenêtre Python Shell :

```
= RESTART: C:/Users/Olivier/Docs/Travail/ArdPyLab/Docs/Python/Invent-pickle.py =
////////////////// INVENTAIRE MATERIEL //////////////////
appuyer sur A pour ajouter un matériel à l'inventaire.
appuyer sur S pour supprimer un matériel de l'inventaire.
appuyer sur V pour afficher la liste de matériel.
appuyer sur Q pour quitter.
////////////////////////////////////////////////////////

v
[Errno 2] No such file or directory: 'fileInvent'
a
saisissez le type de matériel à ajouter à l'inventaire:pipette
saisissez la quantité de ce matériel:20
v
pipette : 20
a
saisissez le type de matériel à ajouter à l'inventaire:becher
saisissez la quantité de ce matériel:10
v
pipette : 20
becher : 10
s
saisissez l'élément à supprimer dans l'inventaire:becher
v
pipette : 20
q
Fin du programme
>>>
```

. Installation de bibliothèques (package) Python

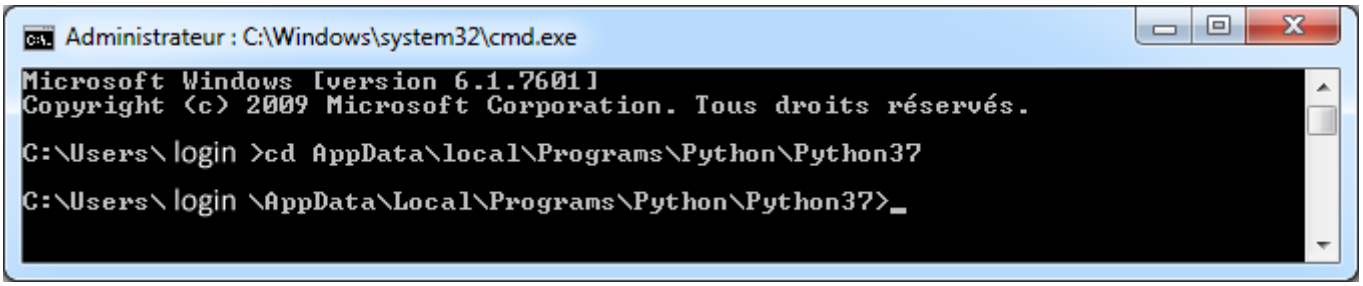
Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines.
Le site pypi.python.org/pypi (The Python Package Index) recense des milliers de modules et de packages !

Nous allons nous intéresser plus particulièrement aux bibliothèques **numpy** et **matplotlib**.

En effet, dans le domaine scientifique, la manipulation des données à travers des tableaux et le tracé de courbes caractéristiques est courante, et les bibliothèques **numpy** et **matplotlib** sont très utiles pour l'exploitation de ces données.

numpy et **matplotlib** ne font pas partie des bibliothèques standard de Python. Il faut donc ajouter ces bibliothèques à la distribution de Python.
A noter, que l'installation de **matplotlib** installe également la bibliothèque de calcul **numpy.**

Pour ajouter une bibliothèque à Python sous Windows, le plus simple est d'ouvrir une console de commandes (taper **cmd** dans la zone de recherche de Windows) et de se placer dans le dossier d'installation de Python :



(A adaptez bien-sûr à votre chemin d'installation, login et n° de version Python…)

Et de taper la commande suivante qui va installer la dernière version d'un package et de ses dépendances depuis le *Python Package Index* (**pip**) qui est l'outil d'installation de prédilection des bibliothèques (à partir de Python 3.4, il est inclus par défaut avec l'installateur de Python) :

```
python -m pip install package
```

Si un package est déjà installé, l'installer à nouveau n'aura aucun effet. La mise à jour de package existants doit être demandée explicitement :

```
python -m pip install --upgrade package
```

Donc, pour installer la bibliothèque **matplotlib**, il suffit de taper la ligne de commande suivante :

```
python -m pip install matplotlib
```

Et l'outil **pip** va télécharger sur le site de dépôts la librairie demandée ainsi que les dépendances nécessaires dont **numpy**.

. numpy

numPy (diminutif de numerical Python) est la bibliothèque indispensable pour le calcul scientifique avec Python.
Cette bibliothèque est utile pour manipuler des matrices ou tableaux multidimensionnels ainsi que les fonctions mathématiques opérant sur ces tableaux.

Voyons les bases de l'utilisation de numpy :

Il faut au départ importer le package numpy avec l'instruction recommandée suivante :

```
>>> import numpy as np
>>>
```

Toutes les fonctions de **NumPy** seront alors préfixées par **np**.

Le package **NumPy** permet la manipulation simple et efficace des tableaux en ajoutant à Python le type **array** similaire à une liste (type **list**). Mais contrairement aux listes, les tableaux **Numpy** ne peuvent contenir que des membres d'un seul type.

Pour créer un tableau Numpy, on peut convertir une liste avec la fonction **array()**:

```
>>> a= np.array([1, 2, 3, 4], int)
>>> a
array([1, 2, 3, 4])
>>> type(a)
<class 'numpy.ndarray'>
```

Le deuxième argument est optionnel et spécifie le type des éléments du tableau :

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([1., 4., 5., 8.])
```

Si dans la liste de départ, il y a des données de types différents, **Numpy** essaiera de les convertir toutes au type le plus général. Par exemple, les entiers **int** seront convertis en nombres à virgule flottante **float** :

```
>>> a = np.array([3.14, 4, 2, 3])
>>> a
array([3.14, 4.  , 2.  , 3.  ])
```

Un tableau peut être multidimensionnel ; ici 2 dimensions :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

Comme pour les listes, on peut accéder aux éléments d'un tableau (attention, comme pour les listes, les indices des éléments commencent à zéro) :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a[0,2]
3
>>> a[1,1]
5
```

Et le slicing (découpage) extrait les tableaux :

```
>>> a = np.array([0,1,2,3,4,5,6,7,8,9])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [début:fin:pas]
array([2, 5, 8])
>>> a[2:8:3] # le dernier élément n'est pas inclus
array([2, 5])
>>> a[:5]    # le dernier élément n'est pas inclus
array([0, 1, 2, 3, 4])
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])  # le dernier élément est inclus
>>> a[2:9]
array([2, 3, 4, 5, 6, 7, 8]) # le dernier élément n'est pas inclus
```

Dans l'instruction **[début:fin:pas]**, deux des arguments peuvent être omis : par défaut l'indice de début vaut 0 (le 1er élément du tableau), et le pas vaut 1.

Un pas négatif inversera l'ordre du tableau :

```
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>> a[:2:-1]
array([9, 8, 7, 6, 5, 4, 3])
```

Et avec un début négatif, la lecture commence par la fin :

```
>>> a[-1::1]
array([9])
```

Pour un tableau bi-dimensionnel, on peut bien-sûr travailler avec les deux indices :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a[:1,:2]
array([[1, 2]])
>>> a[-1,:2]
array([4, 5])
```

Et on peut modifier les valeurs d'un tableau :

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a[:1,:2]=7
>>> a
array([[7, 7, 3],
       [4, 5, 6]])
```

En fait, un tableau multidimensionnel est représenté par une liste de listes, et au final, un tableau bi-dimensionnel (lignes et colonnes) n'est rien d'autre qu'une liste de lignes, une ligne étant une liste de nombres.

On peut alors facilement créer un tableau bi-dimensionnel avec la fonction **range()** :

```
>>> a = np.array([range(i, i + 10) for i in [0, 10]])
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

Ce qui donne le tableau suivant :

	Col 0	**Col 1**	**Col 2**	**Col 3**	**Col 4**	**Col 5**	**Col 6**	**Col 7**	**Col 8**	**Col 9**
Ligne 0 a[0]	0	1	2	3	4	5	6	7	8	9
Ligne 1 a[1]	10	11	12	13	14	15	16	17	18	19

Avec :

```
>>> a = np.array([range(i, i + 10) for i in [0, 10]])
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
>>> a[0]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1]
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> a[0,3]
3
>>> a[1,5]
15
```

Cependant, **Numpy** dispose de plusieurs fonctions pour créer directement des tableaux :

. Zeros()

```
# Un tableau bi-dimensionnel de taille 1x10, rempli d'entiers qui valent 0
>>> np.zeros(10, int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

. Ones()

```
# Un tableau bi-dimensionnel de taille 3x5, rempli de nombres à virgule flottante de valeur 1
>>> np.ones((3, 5), float)
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

. full()

```
# Un tableau bi-dimensionnel de taille 3x5, rempli de 2
>>> np.full((3, 5), 2)
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])
```

. arange()

```
# Un tableau bi-dimensionnel de taille 1x10, rempli d'une séquence linéaire d'entiers
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# Un tableau rempli d'une séquence linéaire de nombres à virgule flottante
>>> np.arange(2, 10, dtype=np.float)
array([2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(3.0)
array([0., 1., 2.])
# Un tableau rempli d'une séquence linéaire d'entiers par pas de 2
>>> np.arange(0, 20, 2)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
# Un tableau rempli d'une séquence linéaire de nombres à virgule flottante par pas de 0.1
>>> np.arange(2, 3, 0.1)
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

Remarques :

Dans l'instruction np.arange**(début:fin:pas)**, deux des arguments peuvent être omis :

. **début** : par défaut l'indice de début vaut 0 (le 1er élément du tableau)

. **pas** : par défaut le pas vaut 1.

Le dernier élément du tableau est l'argument **fin** auquel il faut retrancher le **pas**.

. linspace()

Comme il y quelques subtilités avec la fonction **arange()** quant au dernier élément, pour éviter tout problème, la fonction **linspace(premier,dernier,n)** renvoie un array commençant par **premier**, se terminant par **dernier** avec **n** éléments.

```
>>> np.linspace(1., 4., 6)
array([1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

. **reshape ()**

La fonction reshape() permet de redimensionner un tableau. Il faut cependant que le nombre d'éléments reste le même.

```
>>> a=np.arange(16)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> a=a.reshape(4,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a=a.reshape(2,8)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
```

NumPy dispose d'un grand nombre de fonctions mathématiques qui peuvent être appliquées directement à un tableau. Dans ce cas, la fonction est appliquée à chacun des éléments du tableau.

```
>>> x= np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y=2*x
>>> y
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
>>> x = np.linspace(-np.pi/2, np.pi/2, 3)
>>> x
array([-1.57079633,  0.        ,  1.57079633])
>>> y = np.sin(x)
>>> y
array([-1.,  0.,  1.])
```

Les fonctions mathématiques couramment utilisées sont :

. **numpy.sin(x)**	sinus
. **numpy.cos(x)**	cosinus
. **numpy.tan(x)**	tangente
. **numpy.arcsin(x)**	arcsinus
. **numpy.arccos(x)**	arccosinus
. **numpy.arctan(x)**	arctangente
. **x**n**	x à la puissance n, exemple : x**2
. **numpy.sqrt(x)**	racine carrée
. **numpy.exp(x)**	exponentielle
. **numpy.log(x)**	logarithme népérien
. **numpy.abs(x)**	valeur absolue

. **numpy.around(x,n)** arrondi à n décimales

On va donc pouvoir appliquer n'importe quelle fonction mathématique à un tableau de données **x** de façon à obtenir la caractéristique **y=f(x)**.

Cette caractéristique pourra être tracé à l'aide de la bibliothèque **matplolib**.

. matplotlib

matplotlib est une bibliothèque destinée à tracer et visualiser des données sous formes de graphiques.

Pour tracer des graphes **y=f(x)** avec les points reliés, où x et y sont fournis par des tableaux **numpy**, il faut importer le module **pyplot** de **matplotlib** (**numpy** étant aussi utilisé, il devra bien-sûr être également importé).

Nous allons commencer par un exemple simple, le tracé une droite d'équation : **y = ax+ b**

plot.py - D:/Travail/ArdPyLab/Docs/Python/plot.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

a = 2
b = 5
x = np.arange(10)
print('x:', x)
y = a*x+b
print('y:', y)

plt.plot(x, y)
plt.show()
```

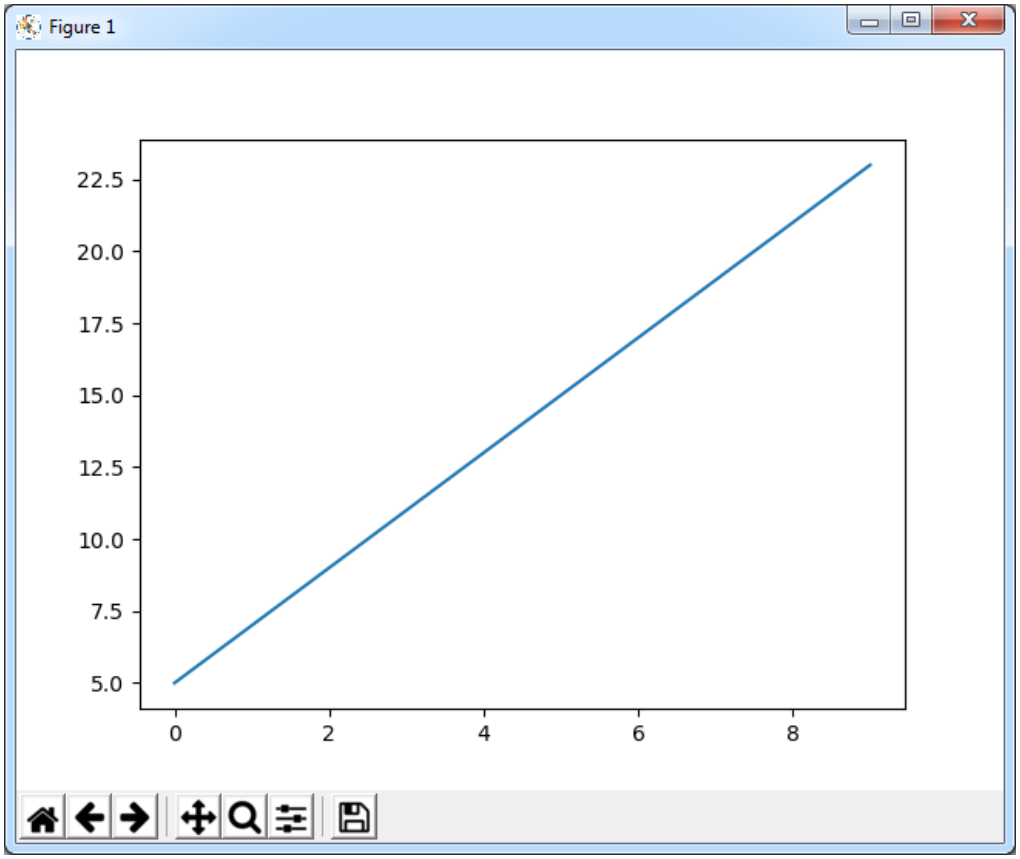
En premier, il faut créer un tableau numpy d'entiers, par exemple de 0 à 9, correspondant aux abscisses du graphe (**x=np.arange(10)**).

On applique la fonction de la droite à ce tableau pour obtenir un autre tableau numpy d'entiers correspondant aux ordonnées du graphes (**y= a*x + b**)

Résultats dans le fenêtre Python Shell :

```
=============== RESTART: D:/Travail/ArdPyLab/Docs/Python/plot.py ===============
x: [0 1 2 3 4 5 6 7 8 9]
y: [ 5  7  9 11 13 15 17 19 21 23]
```

Le graphe est créé avec l'instruction **plt.plot(x,y)** et affiché avec **plt.show()** :



Selon le même principe, nous allons maintenant tracer une sinusoïde :

plot2.py - D:/Travail/ArdPyLab/Docs/Python/plot2.py (3.7.7)

File Edit Format Run Options Window Help

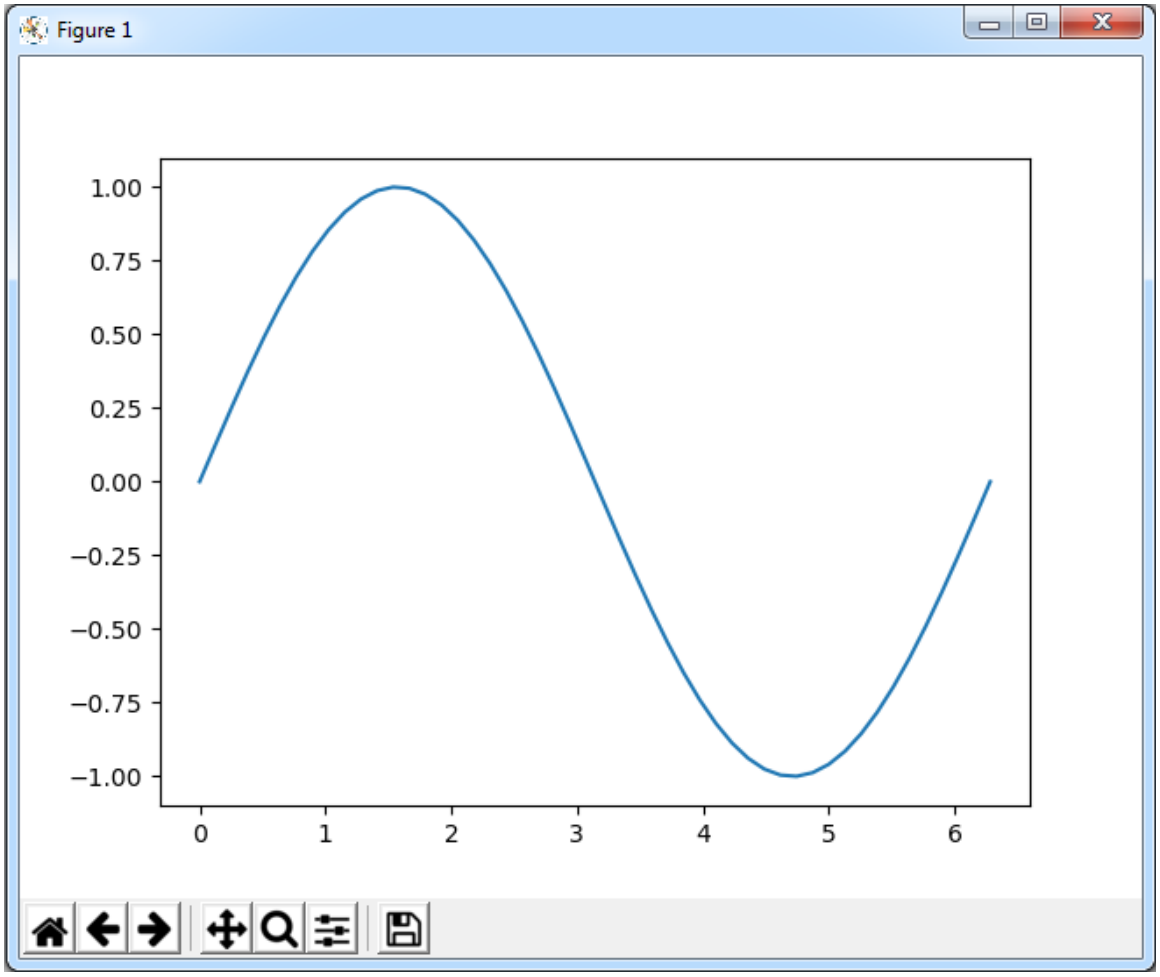
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
print('x:',x)
y = np.sin(x)
print('y:',y)

plt.plot(x, y)
plt.show()
```

Dans cet exemple, le tableau des abscisses contient 50 éléments également espacés de 0 à 2π. On applique à ce tableau, la fonction sinus, pour obtenir le tableau des ordonnées de 50 éléments également (**y=np.sin(x)**).

Résultats dans le fenêtre Python Shell :

```
============== RESTART: D:/Travail/ArdPyLab/Docs/Python/plot2.py ==============
x: [0.         0.12822827 0.25645654 0.38468481 0.51291309 0.64114136
 0.76936963 0.8975979  1.02582617 1.15405444 1.28228272 1.41051099
 1.53873926 1.66696753 1.7951958  1.92342407 2.05165235 2.17988062
 2.30810889 2.43633716 2.56456543 2.6927937  2.82102197 2.94925025
 3.07747852 3.20570679 3.33393506 3.46216333 3.5903916  3.71861988
 3.84684815 3.97507642 4.10330469 4.23153296 4.35976123 4.48798951
 4.61621778 4.74444605 4.87267432 5.00090259 5.12913086 5.25735913
 5.38558741 5.51381568 5.64204395 5.77027222 5.89850049 6.02672876
 6.15495704 6.28318531]
y: [ 0.00000000e+00  1.27877162e-01  2.53654584e-01  3.75267005e-01
  4.90717552e-01  5.98110530e-01  6.95682551e-01  7.81831482e-01
  8.55142763e-01  9.14412623e-01  9.58667853e-01  9.87181783e-01
  9.99486216e-01  9.95379113e-01  9.74927912e-01  9.38468422e-01
  8.86599306e-01  8.20172255e-01  7.40277997e-01  6.48228395e-01
  5.45534901e-01  4.33883739e-01  3.15108218e-01  1.91158629e-01
  6.40702200e-02 -6.40702200e-02 -1.91158629e-01 -3.15108218e-01
 -4.33883739e-01 -5.45534901e-01 -6.48228395e-01 -7.40277997e-01
 -8.20172255e-01 -8.86599306e-01 -9.38468422e-01 -9.74927912e-01
 -9.95379113e-01 -9.99486216e-01 -9.87181783e-01 -9.58667853e-01
 -9.14412623e-01 -8.55142763e-01 -7.81831482e-01 -6.95682551e-01
 -5.98110530e-01 -4.90717552e-01 -3.75267005e-01 -2.53654584e-01
 -1.27877162e-01 -2.44929360e-16]
```

Le graphe est créé avec l'instruction **plt.plot(x,y)** et affiché avec **plt.show()** :



Les graphes ainsi obtenus peuvent être mis en forme :

- Domaine des axes des abscisses et des ordonnées :

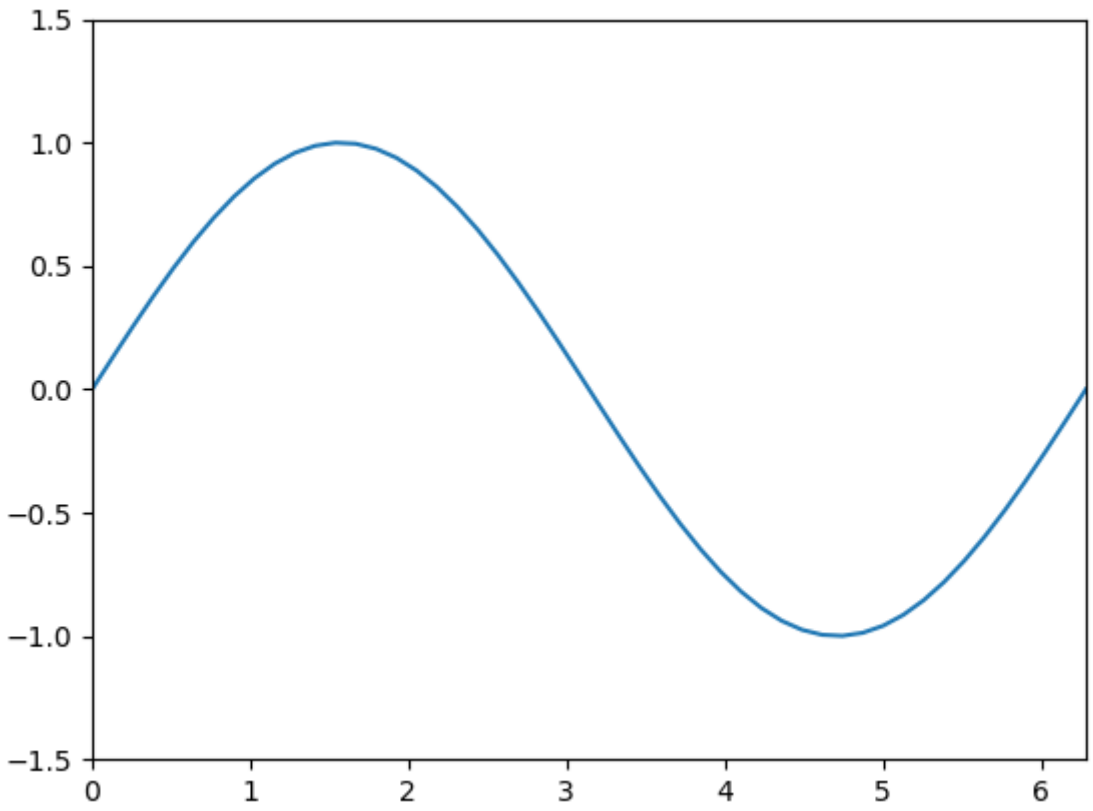
Il est possible fixer indépendamment les domaines des abscisses et des ordonnées en utilisant les fonctions **xlim()** et **ylim()**.

plot3.py - D:/Travail/ArdPyLab/Docs/Python/plot3.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.5, 1.5)

plt.plot(x, y)
plt.show()
```

Ce qui donne :



- Ajout d'un titre :

On peut ajouter un titre grâce à l'instruction **title()** : **plt.title("titre")**

- Ajout d'une légende :

L'instruction **plt.legend()** ajoute la légende au graphe définie lors de la création du graphe :

plt.plot(x,y, "legende")

- Ajout d'étiquettes sur les axes :

Les fonctions **xlabel()** et **ylabel()** ajoutent respectivement des étiquettes sur les axes des abscisses et des ordonnées.

plt.xlabel("abscisses")

plt.ylabel("ordonnees")

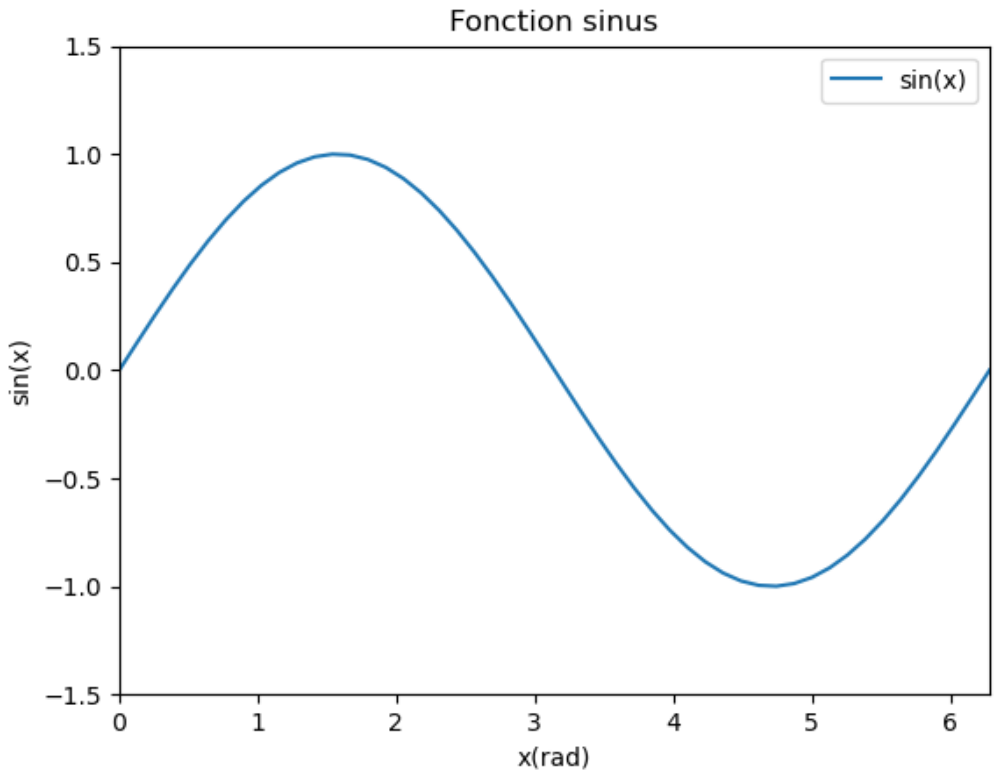
Exemple de mise en forme :

plot4.py - D:/Travail/ArdPyLab/Docs/Python/plot4.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.5, 1.5)

plt.title("Fonction sinus")

plt.xlabel("x(rad)")
plt.ylabel("sin(x)")

plt.plot(x, y, label="sin(x)")

plt.legend()
plt.show()
```

Ce qui donne :



Il est possible d'afficher plusieurs courbes sur le même graphe. Pour chaque courbe, il suffit de définir des tableaux **numpy** correspondant aux valeurs des ordonnées des caractéristiques **y=f(x)**.

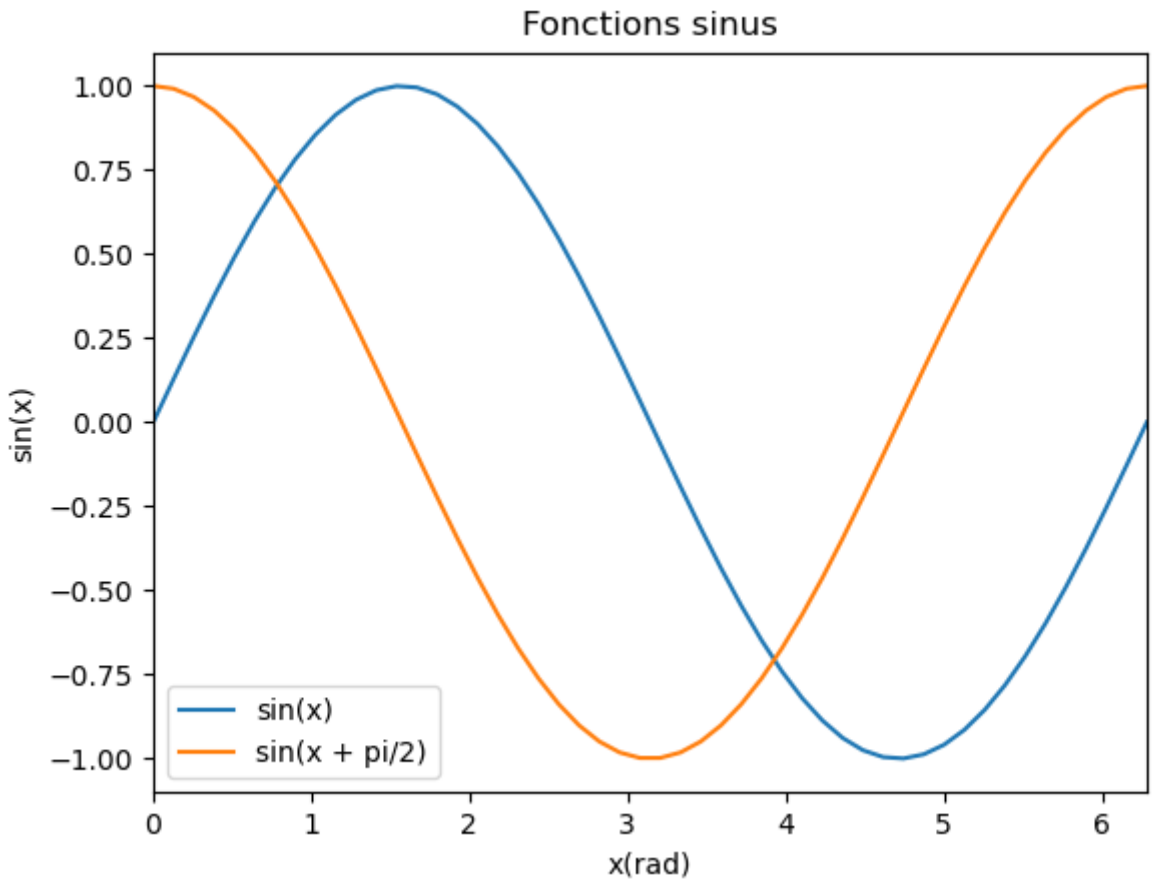
Exemple : Affichage de deux sinusoïdes avec un déphasage de π/2

plot5.py - D:/Travail/ArdPyLab/Docs/Python/plot5.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.sin(x+np.pi/2)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.1, 1.1)

plt.title("Fonctions sinus")

plt.xlabel("x(rad)")
plt.ylabel("sin(x)")

plt.plot(x, y, label="sin(x)")
plt.plot(x, y2, label="sin(x + pi/2)")

plt.legend()
plt.show()
```

Ce qui donne :



-Style des courbes

Le style des courbes est modifiable en précisant la couleur, le style de ligne et les symboles des points ("marker") en ajoutant une chaîne de caractères à l'instruction de création du graphe, de la façon suivante :

plot(x, y, "couleur symbole style de ligne", label="label")

. couleurs

Les chaînes de caractères suivantes permettent de définir la couleur :

Chaîne	Couleur
b	Blue (bleu)
g	Green (vert)
r	Red (rouge)
c	Cyan
m	magenta
y	Yellow (jaune)
k	Black (noir)
w	White (blanc)

. Styles de ligne

Les chaînes de caractères suivantes permettent de définir le style de ligne :

Chaîne	Effet
-	ligne continue
--	tirets
:	ligne en pointillé
-.	tirets points

. Symboles

Les chaînes de caractères suivantes permettent de définir le symbole ("marker") :

Chaîne	Effet	Chaîne	Effet
.	point marker	s	square marker
,	pixel marker	p	pentagon marker
o	circle marker	*	star marker
v	triangle_down	h	hexagon1 marker
^	triangle_up marker	H	hexagon2 marker
<	triangle_left marker	+	plus marker
>	triangle_right	x	x marker
1	tri_down marker	D	diamond marker
2	tri_up marker	d	thin_diamond marker
3	tri_left marker	|	vline marker
4	tri_right marker	_	hline marker

Remarque :

Pour une représentation graphique en nuage de points, il suffit d'indiquer un symbole sans préciser un style de ligne.

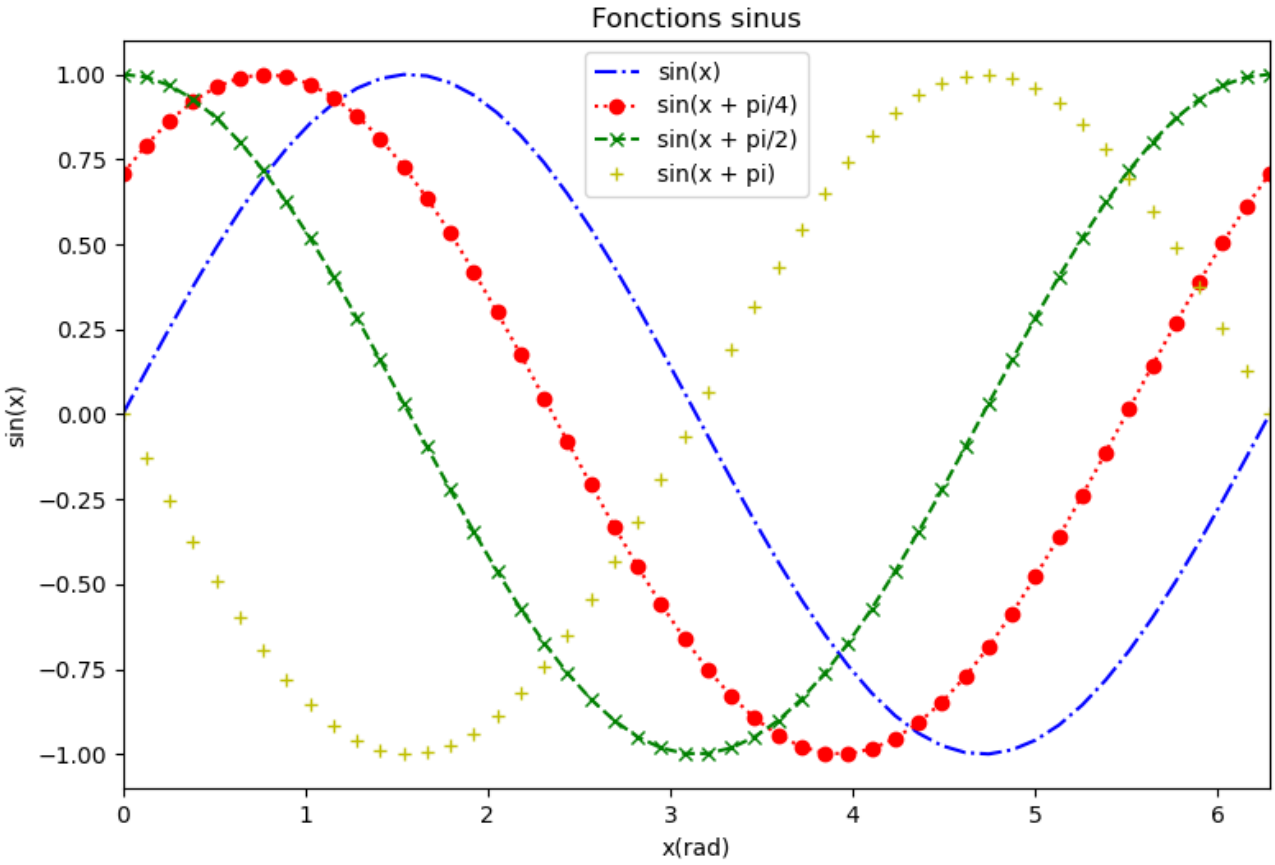
Exemples de styles de courbe :

plot5.py - D:\Travail\ArdPyLab\Docs\Python\plot5.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.sin(x+np.pi/4)
y3 = np.sin(x+np.pi/2)
y4 = np.sin(x+np.pi)

plt.xlim(0, 2*np.pi)
plt.ylim(-1.1, 1.1)

plt.title("Fonctions sinus")

plt.xlabel("x(rad)")
plt.ylabel("sin(x)")

plt.plot(x, y, "b-.",label="sin(x)")
plt.plot(x, y2,"r:o", label="sin(x + pi/4)")
plt.plot(x, y3,"g--x", label="sin(x + pi/2)")
plt.plot(x, y4,"y+", label="sin(x + pi)")

plt.legend()
plt.show()
```

Ce qui donne :

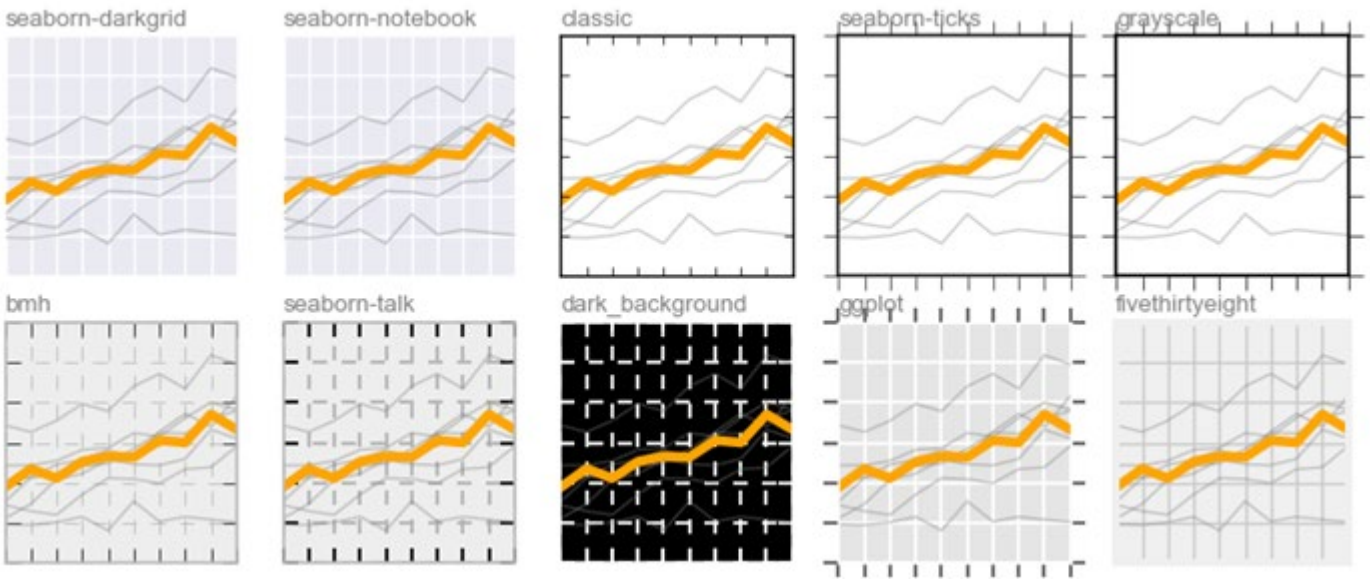


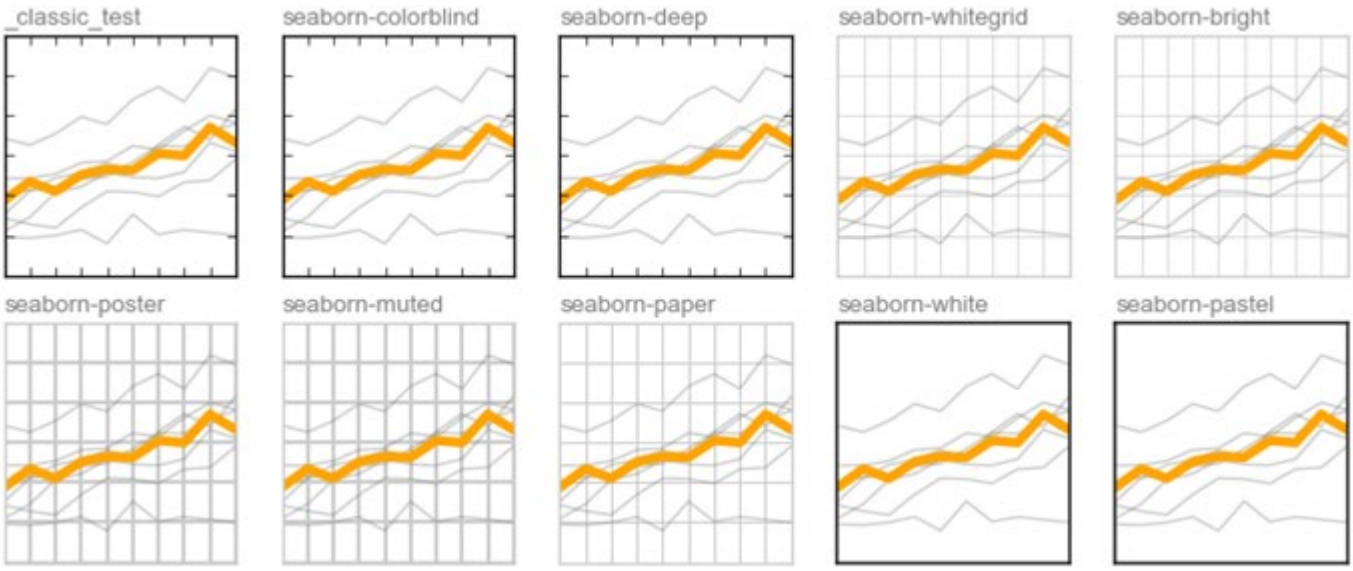
- <u>style de graphe</u> :

Le style du graphe est modifiable avec l'instruction :

pyplot.style.use('nom du style')

Voici quelques styles de graphes de **matplotlib** :





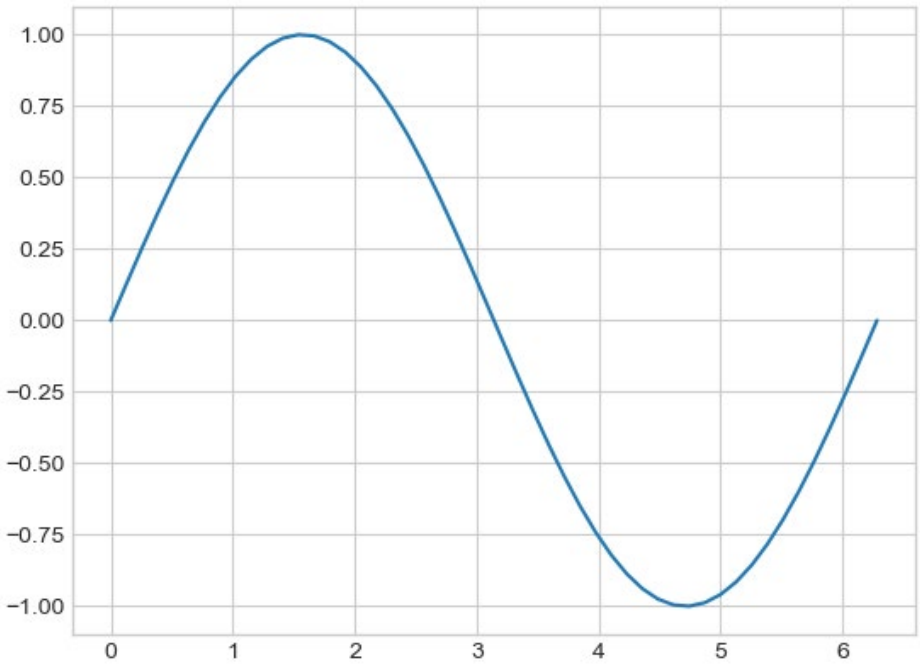
Exemple : Style ‘’seaborn-whitegrid’’

plot6.py - D:/Travail/ArdPyLab/Docs/Python/plot6.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('seaborn-whitegrid')

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)


plt.plot(x, y)
plt.show()
```

Ce qui donne :



- Disposition et graphes multiples :

Il est possible d'afficher plusieurs graphes sur la même figure en créant un objet **figure** dont on peut préciser la taille en pouce :

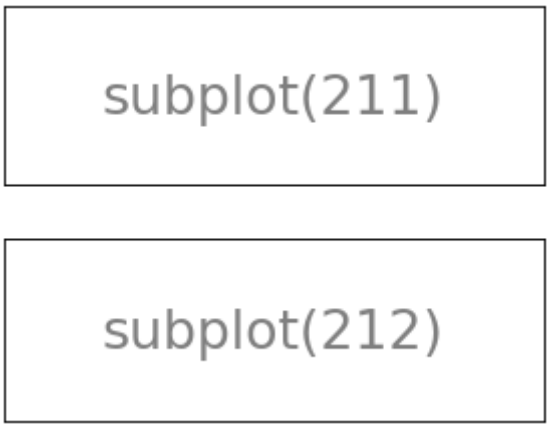
fig = pyplot.figure(figsize = (10, 10))

Les graphes dans cette figure sont modélisés par des objets **axes**. Pour créer un graphe dans la figure **fig,** on crée un objet **axe** à l'aide de la fonction **subplot()** en spécifiant le nombre de lignes et le nombre de colonnes dans la figure, ainsi que le numéro du graphe :

ax1 = pyplot.subplot(211)

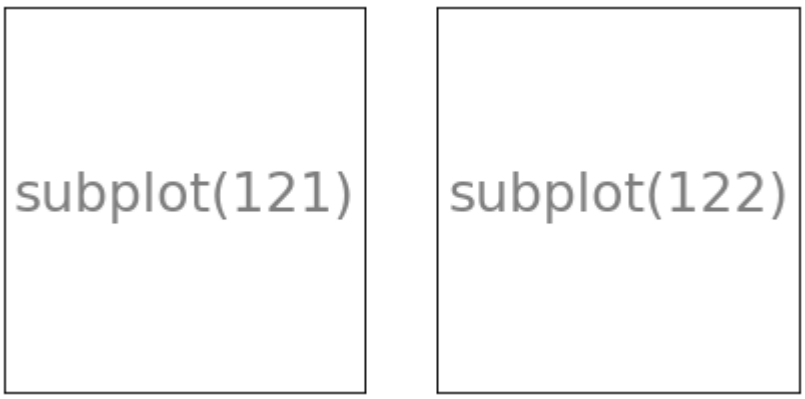
ax2 = pyplot.subplot(212)

Ces instructions vont créer 2 lignes et 1 colonne dans la figure, les 2 lignes contenant chacune 1 graphe :



Ou : **ax1 = plt.subplot(121)**

ax2 = plt.subplot(122)



1 ligne, 2 colonnes

Ou : **ax1 = plt.subplot(221)**

ax2 = plt.subplot(222)

ax3 = plt.subplot(223)

ax4 = plt.subplot(224)

2 lignes, 2 colonnes

Et ainsi de suite…

Les graphes sont créés avec la fonction **plot()** appliquée aux axes définis :

ax1.plot(x, y)

ax2.plot(x, y2)

…

et affichés avec la fonction **show()** appliquée à la figure :

fig.show()

La définition des styles de courbes reste le même :

axe.plot(x, y, "couleur symbole style de ligne", label="label")

et l'affichage de la légende se fait sur l'objet **axe** :

axe.legend()

La mise en forme des graphes (titre, échelles, étiquettes des axes) est cependant légèrement différent :

- Ajout d'un titre sur un objet **axe** :

axe.set_title("titre")

- Définition des échelles des axes :

axe.set_xlim(x_1, x_2)

axe.set_ylim(y_1, y_2)

- Ajout d'étiquettes sur les axes :

axe.set_xlabel("labelx")

axe.set_ylabel("labely")

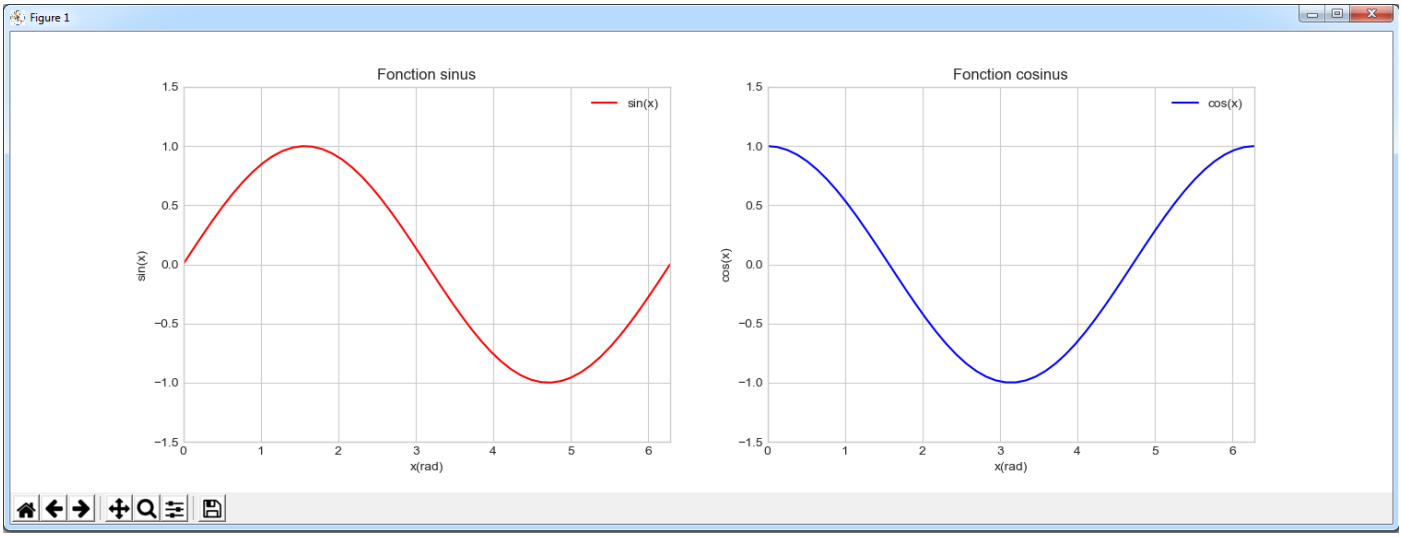
Exemples :

. 2 graphes sur 1 ligne / 2 colonnes :

fig2.py - D:\Travail\ArdPyLab\Docs\Python\fig2.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('seaborn-whitegrid')

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.cos(x)

fig = plt.figure(figsize = (15, 5))
ax1 = plt.subplot(121)
ax2 = plt.subplot(122)

ax1.set_title("Fonction sinus")
ax2.set_title("Fonction cosinus")

ax1.set_xlim(0, 2*np.pi)
ax1.set_ylim(-1.5, 1.5)
ax2.set_xlim(0, 2*np.pi)
ax2.set_ylim(-1.5, 1.5)

ax1.set_xlabel("x(rad)")
ax2.set_xlabel("x(rad)")
ax1.set_ylabel("sin(x)")
ax2.set_ylabel("cos(x)")

ax1.plot(x, y,"r",label="sin(x)")
ax2.plot(x, y2,"b",label="cos(x)")

ax1.legend()
ax2.legend()

fig.show()
```



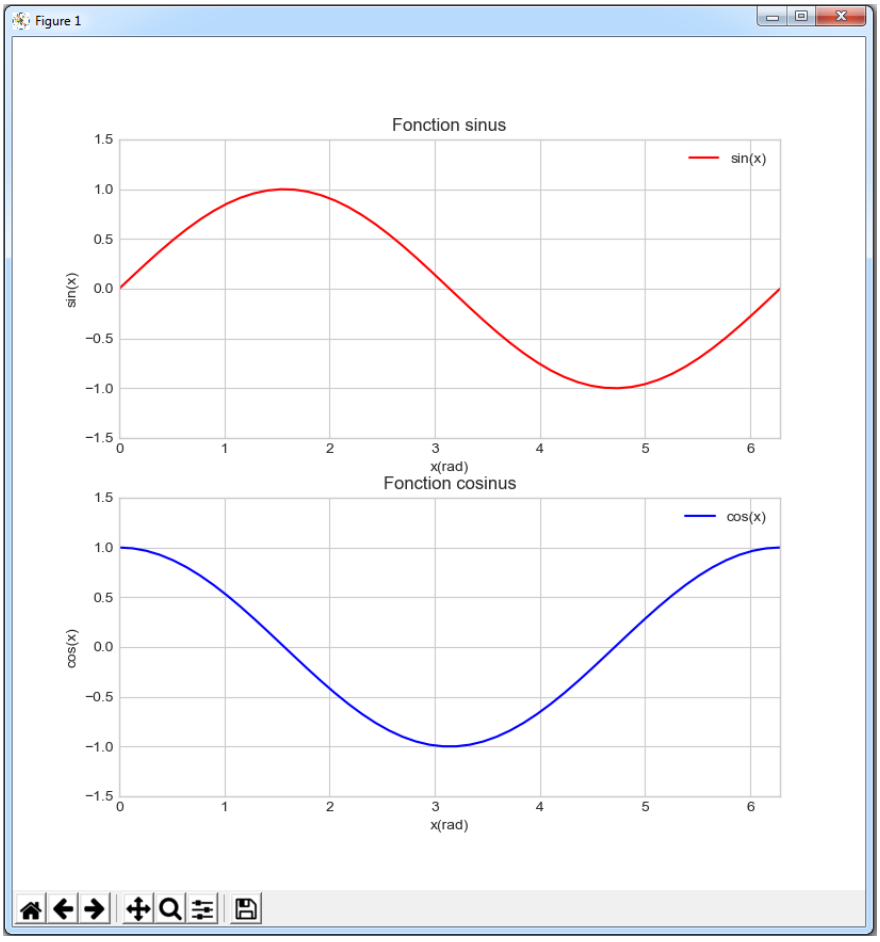
. 2 graphes sur 2 lignes / 1 colonne :

Le code est identique à celui de l'exemple précèdent seul la taille de la figure et la disposition des graphes changent :

fig = plt.figure(figsize = (8, 8))

ax1 = plt.subplot(211)

ax2 = plt.subplot(212)

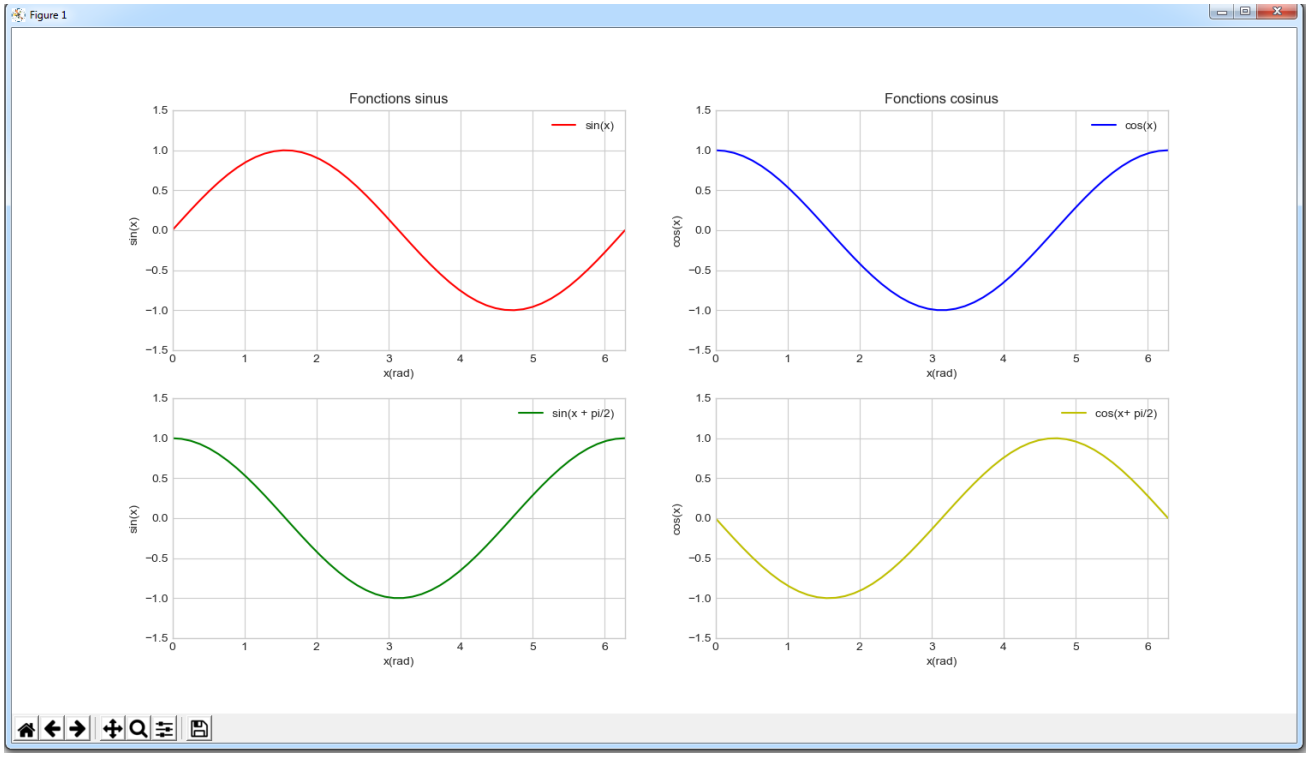


. 4 graphes sur 2 lignes / 2 colonnes :

fig3.py - D:/Travail/ArdPyLab/Docs/Python/fig3.py (3.7.7)

File Edit Format Run Options Window Help

```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('seaborn-whitegrid')

x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = np.cos(x)
y3 = np.sin(x+np.pi/2)
y4 = np.cos(x+np.pi/2)

fig = plt.figure(figsize = (15, 8))
ax1 = plt.subplot(221)
ax2 = plt.subplot(222)
ax3 = plt.subplot(223)
ax4 = plt.subplot(224)

ax1.set_title("Fonctions sinus"), ax2.set_title("Fonctions cosinus")

ax1.set_xlim(0, 2*np.pi),ax1.set_ylim(-1.5, 1.5)
ax2.set_xlim(0, 2*np.pi),ax2.set_ylim(-1.5, 1.5)
ax3.set_xlim(0, 2*np.pi),ax3.set_ylim(-1.5, 1.5)
ax4.set_xlim(0, 2*np.pi),ax4.set_ylim(-1.5, 1.5)

ax1.set_xlabel("x(rad)"),ax2.set_xlabel("x(rad)")
ax3.set_xlabel("x(rad)"),ax4.set_xlabel("x(rad)")
ax1.set_ylabel("sin(x)"),ax3.set_ylabel("sin(x)")
ax2.set_ylabel("cos(x)"),ax4.set_ylabel("cos(x)")

ax1.plot(x, y,"r",label="sin(x)"),ax2.plot(x, y2,"b",label="cos(x)")
ax3.plot(x, y3,"g",label="sin(x + pi/2)"),ax4.plot(x, y4,"y",label="cos(x+ pi/2)")

ax1.legend(),ax2.legend(),ax3.legend(),ax4.legend()

fig.show()
```



Tout ce qui vient d'être vu n'est qu'une infime partie des possibilités que peut offrir **matplotlib**. Pour plus d'informations sur **matplotlib**, de nombreux tutoriels sont disponibles sur le site :

https://matplotlib.org/3.1.1/tutorials/

3.4.5 La programmation orientée objet (classes et objets)

La programmation orientée objet (ou POO en abrégé) est une autre manière de construire et d'organiser son code.

Python est un langage orienté objet, ce qui signifie que le langage tout entier est construit autour de la notion d'objets.

Par exemple, les types **str**, **int**, **list**,... sont des objets, les fonctions sont des objets, etc…

La programmation orientée objet repose sur le concept d'objets qui sont des blocs de code qui possède un ensemble de variables (appelées **attributs** en python) et de fonctions qui leur sont propres (appelées **méthodes** en python).

Les objets sont créés à partir de "modèles" appelés **classes** qui sont également des ensembles de codes qui contiennent des variables et des fonctions. Les objets créés possèdent alors un même ensemble d'attributs et de méthodes que les classes, les attributs étant des variables accessibles depuis toute méthode de la classe où elles sont définies.

On dit qu'un objet est une instance d'une classe. On peut créer autant d'objets que l'on désire avec une classe.

. Les classes

Une classe est équivalente à un type de données et créer une nouvelle classe en Python correspond à définir un nouveau type de données.

Sans le savoir, nous avons déjà vu des classes :

- le type de donnée **str** est une classe dont l'objet **ch = "chaine "** est une instance et par exemple, la fonction **upper()** est une de ses méthodes,

- le type de donnée **list** est une classe dont l'objet **liste=[1,2,3,4]** est une instance et par exemple, la fonction **sort()** est une de ses méthodes,

- …

Pour créer une nouvelle classe Python, on utilise le mot clef **class** suivi du nom de la classe.

Nous allons créer une classe nommée **inventaire** à partir du programme d'inventaire déjà vu en tant qu'exemple d'application de la manipulation des fichiers.

```
class inventaire:
    def __init__(self):
        self.invent={}
        self.inventpath=""
        self.finprog=False
```

Les instructions ci-dessus crée une classe nommée **inventaire** dont les attributs sont :

. un dictionnaire nommé **self.invent** initialement vide,

. une chaine de caractères nommée **self.inventpath** correspondant au chemin de l'inventaire initialement vide,

. une variable booléenne nommée **self.finprog** initialisée en **False** afin de pouvoir mettre fin à l'exécution du programme.

La méthode **_init_(self)** dans laquelle les attributs sont définis est appelée lors de la création d'un objet à partir de la classe. Cette méthode est nommée un **constructeur**.

Pour créer un objet à partir de la classe **inventaire**, il suffira dans le programme principal d'appeler la classe à l'aide de l'instruction : **nom_objet = inventaire()**

La méthode **_init_(self)** est alors appelée et les variables du constructeur sont attribuées à l'objet créé.

La méthode prend en paramètre la variable **self**. Cette variable représente l'objet lui-même (d'où le nom **self**…) et c'est pourquoi les attributs sont de type : **self.variable** (car ce sont les variables de l'objet !).

De cette façon, il n'y a plus besoin de **variables globales** dans des fonctions du programme ayant besoin de modifier des variables externes à la fonction (les attributs **self.variable** étant des variables accessibles depuis toute méthode de la classe).

Après la définition des attributs, il faut définir les méthodes de la classe **inventaire** :

- Méthode pour ouvrir un inventaire :

```
def OuvreInventaire(self):
    self.InventPath=input("Indiquez le nom de l'inventaire:")
    self.invent = {}
    try:
        with open(self.InventPath, 'r') as fichier:
            for line in fichier:
                listline=line.split(";")
                self.invent[listline[0]]=listline[1].strip()

    except Exception as message:
        print(message)
```

Cette méthode demande à l'utilisateur le chemin d'un fichier d'inventaire (**self.InventPath**) et tente de l'ouvrir. Un message est affiché si le fichier n'existe pas, sinon l'inventaire est chargé dans le dictionnaire **self.invent**.

- Méthode pour ajouter un élément à l'inventaire :

```
def AjoutMatos(self):
    Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
    Quant=input("saisissez la quantité de ce matériel:")
    self.invent[Matos]=Quant
```

Cette méthode demande à l'utilisateur de saisir les données (nom et quantité) de l'élément à ajouter à l'inventaire et l'ajoute au dictionnaire **self.invent**.

- Méthode pour effacer un élément de l'inventaire :

```
    def EffaceMatos(self):
        element=input("saisissez l'élément à supprimer dans l'inventaire:")
        try:
            del self.invent[element]
        except:
            print("L'élément que vous voulez supprimer n'existe pas!")
```

Cette méthode demande à l'utilisateur de saisir le nom de l'élément à supprimer de l'inventaire et le supprime du dictionnaire **self.invent**.

- Méthode pour afficher l'inventaire :

```
    def ReadInventaire(self):
        for cle, valeur in self.invent.items():
            print("{} : {}".format(cle, valeur))
```

Cette méthode affiche la liste des éléments de l'inventaire **self.invent** dans la fenêtre Python shell.

- Méthode pour sauvegarder l'inventaire :

```
    def SaveInventaire(self):
        self.InventPath=input("Indiquez le nom de sauvegarde de l'inventaire:")
        with open(self.InventPath, 'w') as fichier:
            for cle, valeur in self.invent.items():
                fichier.write("{};{}".format(cle, valeur))
                fichier.write("\n")
```

Cette méthode demande à l'utilisateur de saisir le chemin d'enregistrement de l'inventaire (**self.InventPath**). L'inventaire est parcouru et sauvegarder dans le fichier ouvert.

- Méthode pour déterminer l'action à exécuter :

```
    def ChoixAction(self):
        print("/////////////// INVENTAIRE MATERIEL ///////////////")
        print("appuyer sur O pour ouvrir un inventaire:")
        print("appuyer sur A pour ajouter un matériel à l'inventaire:")
        print("appuyer sur S pour supprimer un matériel de l'inventaire:")
        print("appuyer sur V pour afficher la liste de matériel:")
        print("appuyer sur E pour enregistrer l'inventaire:")
        print("appuyer sur Q pour quitter:")
        print("////////////////////////////////////////////////")
        print(" ")

        while self.finprog == False:
            choix = " "
            while choix.upper()!="A" or "S" or "Q" or "P" or "O" or "E":
                choix=input()
```

```
            # Action en fonction de l'entrée clavier:
            if choix.upper()=="O": self.OuvreInventaire()
            if choix.upper() == "A": self.AjoutMatos()
            if choix.upper() == "V": self.ReadInventaire()
            if choix.upper() == "S": self.EffaceMatos()
            if choix.upper()=="E": self.SaveInventaire()
            if choix.upper() == "Q":
                print("Fin du programme")
                self.finprog = True
                break
```

Cette méthode affiche la liste des actions disponibles et demande en boucle à l'utilisateur de faire un choix.

Les attributs et les méthodes de la classe **inventaire** ont maintenant tous été définis. Le programme va créer un objet nommé **mon_inventaire** qui est une instance de la classe **inventaire** :

mon_inventaire=inventaire()

Puis on appelle la méthode **ActionChoix()** de l'objet créé pour lancer la boucle des actions :

mon_inventaire.ChoixAction()

Résultats dans le fenêtre Python Shell :

```
============ RESTART: D:\Travail\ArdPyLab\Docs\Python\inventclass.py ===========
/////////////// INVENTAIRE MATERIEL ///////////////
appuyer sur O pour ouvrir un inventaire:
appuyer sur A pour ajouter un matériel à l'inventaire:
appuyer sur S pour supprimer un matériel de l'inventaire:
appuyer sur V pour afficher la liste de matériel:
appuyer sur E pour enregistrer l'inventaire:
appuyer sur Q pour quitter:
////////////////////////////////////////////////

o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
a
saisissez le type de matériel à ajouter à l'inventaire:eprouvette
saisissez la quantité de ce matériel:5
e
Indiquez le nom de sauvegarde de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
eprouvette : 5
o
Indiquez le nom de l'inventaire:invent
v
bécher : 10
eprouvette : 15
o
Indiquez le nom de l'inventaire:invent2
v
bécher : 10
erlenmeyer : 20
eprouvette : 5
q
Fin du programme
```

Voici Le programme complet avec la classe **inventaire** qui pourra bien-sûr se situé dans un module à part qui le cas échéant devra être importé avant utilisation :

inventclass.py - D:\Travail\ArdPyLab\Docs\Python\inventclass.py (3.7.7)

File Edit Format Run Options Window Help

```
class inventaire:
    def __init__(self):
        self.invent={}
        self.inventpath=""
        self.finprog=False

    def OuvreInventaire(self):
        self.InventPath=input("Indiquez le nom de l'inventaire:")
        self.invent = {}
        try:
            with open(self.InventPath, 'r') as fichier:
                for line in fichier:
                    listline=line.split(";")
                    self.invent[listline[0]]=listline[1].strip()

        except Exception as message:
            print(message)

    def AjoutMatos(self):
        Matos=input("saisissez le type de matériel à ajouter à l'inventaire:")
        Quant=input("saisissez la quantité de ce matériel:")
        self.invent[Matos]=Quant

    def EffaceMatos(self):
        element=input("saisissez l'élément à supprimer dans l'inventaire:")
        try:
            del self.invent[element]
        except:
            print("L'élément que vous voulez supprimer n'existe pas!")

    def ReadInventaire(self):
        for cle, valeur in self.invent.items():
            print("{} : {}".format(cle, valeur))

    def SaveInventaire(self):
        self.InventPath=input("Indiquez le nom de sauvegarde de l'inventaire:")
        with open(self.InventPath, 'w') as fichier:
            for cle, valeur in self.invent.items():
                fichier.write("{};{}".format(cle, valeur))
                fichier.write("\n")
    def ChoixAction(self):
        print("/////////////// INVENTAIRE MATERIEL ///////////////")
        print("appuyer sur O pour ouvrir un inventaire:")
        print("appuyer sur A pour ajouter un matériel à l'inventaire:")
        print("appuyer sur S pour supprimer un matériel de l'inventaire:")
        print("appuyer sur V pour afficher la liste de matériel:")
        print("appuyer sur E pour enregistrer l'inventaire:")
        print("appuyer sur Q pour quitter:")
        print("////////////////////////////////////////////////")
        print(" ")

        while self.finprog == False:
            choix = " "
            while choix.upper()!="A" or "S" or "Q" or "P" or "O" or "E":
                choix=input()

                # Action en fonction de l'entrée clavier:
                if choix.upper()=="O": self.OuvreInventaire()
                if choix.upper() == "A": self.AjoutMatos()
                if choix.upper() == "V": self.ReadInventaire()
                if choix.upper() == "S": self.EffaceMatos()
                if choix.upper()=="E": self.SaveInventaire()
                if choix.upper() == "Q":
                    print("Fin du programme")
                    self.finprog = True
                    break

mon_inventaire=inventaire()
mon_inventaire.ChoixAction()
```

Tout ce qui a été vu concernant la programmation en Python ne représente que des bases, mais est néanmoins suffisant pour communiquer entre un Arduino Uno et un programme Python à des fins de contrôle et d'exploitation de données.