

# Informatique sur Systèmes Embarqués

Introduction au langage MicroPython  
sur le microcontrôleur

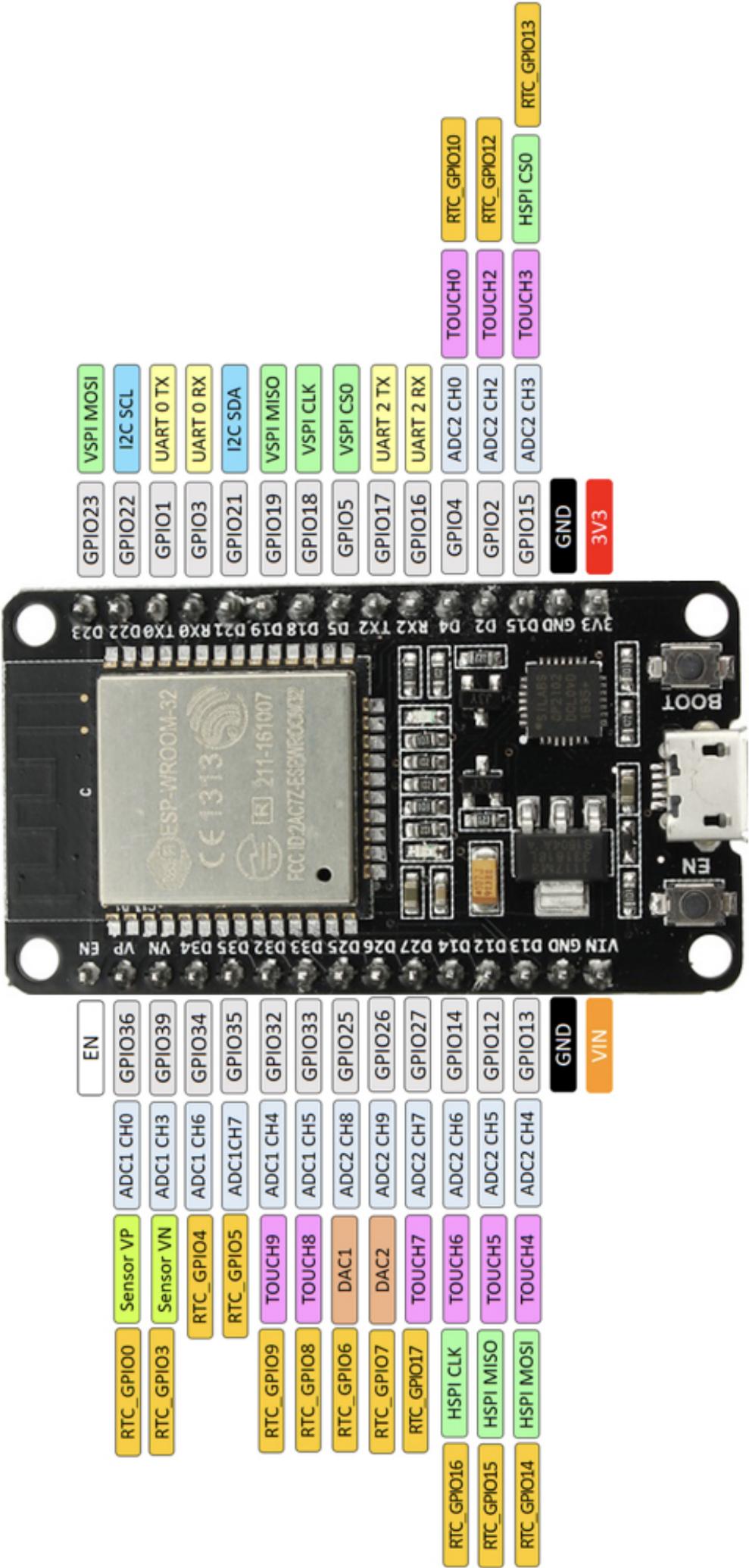
**ESP32**



**ENIM 2020**  
**Vincent HERMITANT**

# ESP32 DEVKIT V1 - DOIT

version with 30 GPIOs



- Informatique sur systèmes embarqués -  
Introduction au langage MicroPython  
sur la base du micro-contrôleur ESP32

Vincent HERMITANT

21 février 2020

## 1 Introduction

### 1.1 Résumé de ce que nous allons apprendre

L'ensemble «ESP32 et MicroPython» permet de piloter (c'est-à-dire commander) tout système mécanique, électronique, technologique, banc d'essai, par un langage de haut niveau. Ce langage est très puissant, largement utilisé dans le monde, et il est interprété sur une carte à microcontrôleur récente (sortie en 2016), puissante, équipée en Wifi et Bluetooth, et d'un coût de moins de 5 euros (en achat direct de Chine). Cette carte à micro-contrôleur est placée au cœur d'un écosystème de capteurs et actionneurs équipant l'installation, le prototype, ou le système.

Ce support de travaux pratiques a pour objectif de vous apprendre à programmer en MicroPython les systèmes embarqués, et en particulier l'ESP32. Nous apprendrons ce qu'est un microcontrôleur, comment le programmer, comment l'implémenter dans son système à piloter, et quels sont les composants à connecter autour. Nous commencerons par comprendre ce que signifie commander un système.

Nul besoin de connaître le langage Python, il sera fait une introduction sur les concepts de base, l'utilisation et création des fonctions, les structures de contrôle conditionnel et les boucles, la notion de classe et d'objet.

### 1.2 Déroulement des séances de Travaux Pratiques

- Séance 1 : Support de TP
- Séance 2 : Support de TP
- Séance 3 : Support de TP + discussion sur le projet (15 min)
- Séance 4 : Support de TP + discussion sur le projet (15 min)
- Séance 5 : Support de TP + interro de cours (30 min) + début du projet (1h)
- Séance 6 : Support de TP + interro de TP (45 min) + suite du projet (1h)
- En dehors des séances de TP : Suite du projet sur 4h
- En dehors des séances de TP : Soutenance du projet

### 1.3 Utilisation de ce support de TP

Tout au long de ce support de TP, de nombreux liens sont disponibles, il sont mis en évidence par une couleur bleue. Ils amènent vers des pages à lire pour de plus d'informations, ou alors vers un dépôt Github qui permet de télécharger les codes cités en exemple.

Pour que les liens de ce support fonctionnent, il est nécessaire de le télécharger, et non pas de l'utiliser en ligne dans une page web.

Par exemple, cliquer sur le lien «test-multiple.py» de la liste des codes ci-dessous. Vous arrivez alors sur mon dépôt Github, et il ne reste plus qu'à cliquer sur RAW pour avoir la version brute du code, puis CTRL+A pour tout sélectionner, puis CTRL+C pour coller, puis CTRL+V quand nous serons dans l'éditeur de script Python pour l'exécuter.

### 1.4 Lien vers les principaux sites

[Lien vers mon dépôt Github : https://github.com/vincent-herm/ESP32-Les-bases](https://github.com/vincent-herm/ESP32-Les-bases)  
[Lien vers le site officiel de MicroPython](#)

Téléchargement de la dernière version du Firmware du ESP32

Librairies de MicroPython : [docs.MicroPython.org/en/latest/library/index.html](https://docs.MicroPython.org/en/latest/library/index.html)

Le langage MicroPython : [docs.MicroPython.org/en/latest/reference/index.html](https://docs.MicroPython.org/en/latest/reference/index.html)

Référence rapide pour ESP32 : [docs.MicroPython.org/en/latest/esp32/quickref.html](https://docs.MicroPython.org/en/latest/esp32/quickref.html)

Lien concernant les interruptions

## Liste des codes mis en œuvre

1	<a href="#">test-multiple.py</a> - Test si n est multiple de x . . . . .	20
2	<a href="#">test-multiple-gestion-erreur.py</a> - Test si n est multiple de x, et gestion erreur . . . . .	21
3	<a href="#">liste-append.py</a> - Crédation d'une liste par ajout d'éléments avec la méthode «append» . . . . .	21
4	<a href="#">pgcd.py</a> - Exemple du calcul du PGCD de deux nombres . . . . .	22
5	<a href="#">classe-1.py</a> - Première approche des classes . . . . .	23
6	<a href="#">classe-2.py</a> - Deuxième approche des classes . . . . .	24
7	<a href="#">classe-3.py</a> - Troisième approche des classes . . . . .	24
8	<a href="#">classe-4.py</a> - Quatrième approche des classes . . . . .	25
9	<a href="#">classe-5.py</a> - Cinquième approche des classes . . . . .	25
10	<a href="#">led.py</a> - Fait clignoter une LED à 1 Hz ; solution 1 . . . . .	29
11	<a href="#">led-3-fois.py</a> - Fait clignoter 3 fois une LED, puis s'arrête . . . . .	29
12	<a href="#">led-temps.py</a> - Fait clignoter 10 000 fois une LED, mesure le temps . . . . .	30
13	<a href="#">led-bouton.py</a> - Allume une LED si un bouton est pressé . . . . .	31
14	<a href="#">led-stop.py</a> - Fait clignoter une LED à 1 Hz avec STOP par bouton . . . . .	31
15	<a href="#">tempo-eclairage.py</a> - Temporisateur d'éclairage . . . . .	32
16	<a href="#">tempo-eclairage-print.py</a> - Temporisateur d'éclairage avec affichage des valeurs . . . . .	32
17	<a href="#">tempo-eclairage-optimise.py</a> - Temporisateur d'éclairage optimisé . . . . .	33
18	<a href="#">pwm.py</a> - PWM - 3 cycles de montée . . . . .	35
19	<a href="#">calcul-correction.py</a> - Calcul correction luminosité . . . . .	35
20	<a href="#">pwm-correction.py</a> - PWM avec correction luminosité . . . . .	36
21	<a href="#">pwm-correction-input.py</a> - PWM avec correction luminosité choisie . . . . .	36
22	<a href="#">pwm-fonction-correction.py</a> - PWM avec correction luminosité par fonction . . . . .	37
23	<a href="#">neopixel.py</a> - Test du ruban NeoPixel . . . . .	37
24	<a href="#">neopixel-test.py</a> - Test du ruban NeoPixel - Deuxième essai . . . . .	38
25	<a href="#">neopixel-deflement.py</a> - Défilement des LED . . . . .	38
26	<a href="#">neopixel-x-allumes.py</a> - L'utilisateur choisit d'allumer x LED . . . . .	39
27	<a href="#">neopixel-random.py</a> - On allume x LED de manière aléatoire . . . . .	39
28	<a href="#">tirage-aleatoire.py</a> - Tirage de 20 valeurs entières entre 0 et 8 . . . . .	40
29	<a href="#">neopixel-rainbow.py</a> - Arc-en-ciel sur NéoPixel . . . . .	40

## 1.5 Commander un système

### 1.5.1 Point commun entre tous les systèmes/objets

Quel est le point commun entre une trottinette électrique, une chaudière d'appartement, une plaque à induction programmable, un radio réveil, et une voiture ?

Réponse : ils sont tous pilotés par un système électronique/informatique, c'est à dire qu'ils contiennent un système de commande, que nous pouvons appeler «contrôleur».

Tous les objets, systèmes, machines, de ce monde sont pilotés ; cela couvre les domaines et objets aussi divers que la machine à café, la serrure à carte de l'hôtel, un drone, une station météo en mer ou un ballon sonde, ...

Ce contrôleur effectue plusieurs opérations :

- Il reçoit des informations de type numérique, ou analogique. Pour cela, un ensemble de capteurs du système à commander est connecté sur ses broches d'entrée,
- Il fournit en réponse des informations destinés à piloter des actionneurs ou toute sorte de système en sortie (LED, relais, transistor, ...), l'information pouvant être sous format analogique, ou numérique.
- Il peut interagir avec un opérateur par le biais d'une IHM (Interface Homme / Machine), par exemple un clavier, écran, bouton, potentiomètres de réglage, ...

— Il peut être connecté via différents réseaux au monde extérieur.

Ainsi, le système de commande est un ensemble électronique/informatique qui est intégré dans une machine mécanique, électrique, et tout objet ou système qui a besoin d'être commandé, régulé, et qui a besoin d'interagir avec son environnement par le biais de capteurs.

### 1.5.2 Système de contrôle embarqué - systèmes embarqués

Les systèmes embarqués sont de plus en plus présents dans notre société où la technologie est omniprésente. Ces systèmes embarqués concernent souvent des applications souvent critiques et temps-réel, et sont soumis à diverses contraintes non fonctionnelles comme l'occupation mémoire ou la consommation d'énergie.

Ils sont souvent «autonomes» et «intelligents», et doivent répondre également à d'autres problématiques comme la robustesse, la sécurité, la fiabilité et la sûreté de fonctionnement. Avec la généralisation des techniques d'échange de données, les systèmes embarqués sont maintenant «communicants».

## 1.6 Le langage MicroPython et les microcontrôleurs

### 1.6.1 Le microcontrôleur

Un microcontrôleur est un circuit intégré qui rassemble en un seul circuit (ou puce) les éléments essentiels d'un ordinateur : processeur, mémoires, unités périphériques et interfaces d'entrées-sorties (GPIO : Global Purpose Input Output). Il réunit toutes les fonctions requises pour lire des informations, traiter des données, et écrire en retour des informations vers le monde physique.

Ainsi, les microcontrôleurs sont fréquemment utilisés dans la gestion des moteurs automobiles, les télécommandes, les appareils de bureau, récepteurs GPS, l'électroménager, les jouets, la téléphonie mobile, etc...

Mais par rapport aux microprocesseurs polyvalents utilisés dans les ordinateurs personnels, les microcontrôleurs se caractérisent par un moins haut degré d'intégration, une plus faible consommation électrique, une vitesse de fonctionnement plus faible, (de quelques MHz jusqu'à environ un GHz), une mémoire assez limitée, et un coût réduit. Ils sont ainsi beaucoup moins puissants. A titre de comparaison, la fréquence d'horloge d'un ordinateur en 2020 est d'environ 3 GHz.

Les microcontrôleurs sont fréquemment utilisés dans les systèmes embarqués.

### 1.6.2 Le langage MicroPython

Le langage MicroPython est une réécriture de Python 3, destinée aux microcontrôleurs. Le tour de force des développeurs a été de réussir à rendre utilisable un langage aussi puissant que Python dans un environnement aussi limité que le microcontrôleur. A part quelques exceptions, les fonctions disponibles dans Python le sont aussi dans MicroPython.

Mais MicroPython n'est pas pourvu de la librairie complète de Python, mais seulement une portion bien choisie, ce qui lui permet d'être intégré et de fonctionner sous ces contraintes matérielles sévères, et un espace mémoire réduit.

Par contre, MicroPython contient en plus les librairies qui permettent l'accès matériel, de bas niveau, pour interagir facilement avec les entrées/sorties, et tout ce qui concerne la gestion de la wifi, et des connections réseau.

Ainsi, MicroPython offre au microcontrôleur des services assez proches d'un système d'exploitation, car il prend en charge le stockage et le transfert de fichiers dans la mémoire Flash, le support de la connexion Wifi-Ethernet, et autres services.

### 1.6.3 Le langage MicroPython associé

L'avantage d'associer MicroPython et l'ESP32 (par exemple) est de rendre l'électronique numérique aussi simple que possible. La démarche est rendue tellement simple qu'il suffit de se former 2 jours environ sur MicroPython et l'ESP32, et l'on est alors capable de développer tout seul des applications qui vous auraient semblé magiques quelques heures auparavant ; seules quelques bases d'électronique et de programmation sont utiles.

Une autre plateforme a eu et a toujours eu beaucoup de passionnés, pour son aspect pratique et Open Source, c'est ARDUINO. Mais aujourd'hui, la puissance de la plateforme à base de MicroPython est telle que ARDUINO pourrait perdre de son aura. En effet, l'ESP32 est beaucoup plus puissant, interprète un

code MicroPython beaucoup pratique que le C de l'ARDUINO à prendre en main. Les cas pour lesquels coder en langage C reste préférable sont très limités.

C'est pour cette raison que les travaux pratique à l'ENIM sont réalisés aujourd'hui sur l'ESP32 alors qu'ils étaient en 2019 réalisés sur ARDUINO.

#### 1.6.4 Exemple de code MicroPython

Exemple de code pour faire clignoter une LED, en la connectant sur une sortie numérique.

```
1 from machine import Pin
2 from time import sleep
3 led = Pin(2, Pin.OUT)
4 while True:
5     led.value(not led.value())
6     sleep(0.5)
```

Mais soulevons deux questions :

- Pourquoi utiliser le langage Python / MicroPython ?
- Pourquoi vouloir utiliser des appareils si petits et si peu puissants ?

#### 1.6.5 Pourquoi utiliser le langage Python ?

Python est un langage de programmation informatique très puissant, facile d'accès, très répandu, et très expressif ; c'est à dire qu'en Python, on écrit du code qui reste très facile à lire, comme un langage naturel (anglais). De plus, Python tient sa vaste popularité à sa vaste communauté bien organisée et très active. En 2016, Python était le troisième langage le plus utilisé dans le monde, derrière le C, et le Java. et prend la place de leader en 2017. Depuis, [Python conforte sa place de leader en 2018 selon l'IEEE](#), la plus grande organisation mondiale de professionnels pour le développement de la technologie. En effet, Python qui a pris la tête du classement général des langages de programmation, dépassant Java et C, sur une bonne partie des critères suivants : popularité générale, langages en forte croissance, langages les plus demandés par les employeurs, les meilleurs langages pour le développement de sites et applications web, pour le développement d'applications mobiles, pour le développement d'applications d'entreprise, de bureau et scientifiques, et pour le développement de systèmes embarqués.

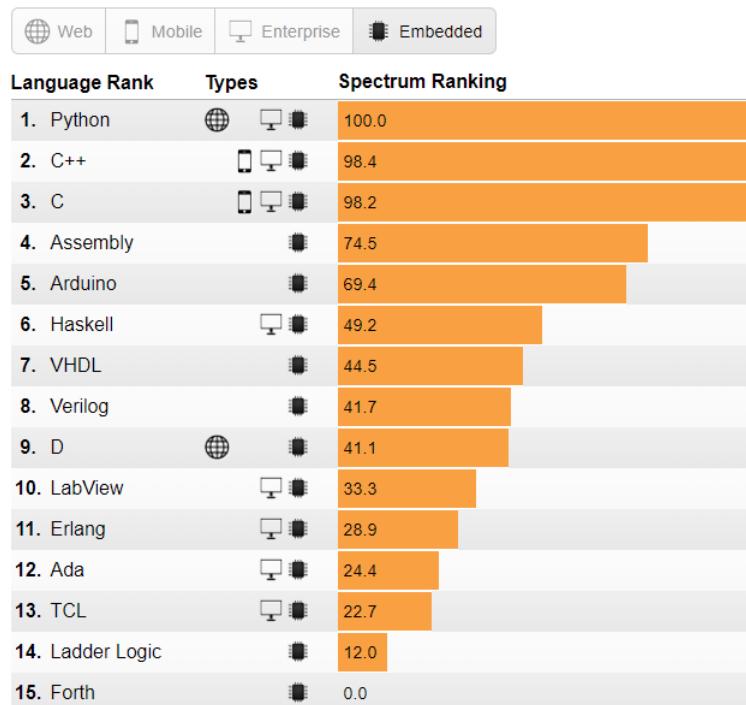


FIGURE 1 – Meilleurs langages pour le développement de systèmes embarqués (selon IEEE en 2018)

Toujours d'après ce même article, et toujours d'après [l'IEEE](#), la popularité de Python « s'explique d'abord par le fait que Python est maintenant répertorié en tant que langage pour l'embarqué. L'écriture d'applications embarquées était la chasse gardée des langages compilés (comme le C, le C++, ...),

parce que cela impliquait des machines avec une puissance de traitement et une mémoire limitées. Mais aujourd’hui, beaucoup de microcontrôleurs modernes ont assez de puissance pour héberger un interpréteur Python. Et un aspect intéressant de l’utilisation de Python dans ce domaine serait qu’il est très pratique dans certaines applications de communiquer avec du matériel via une invite interactive ou de recharger dynamiquement des scripts à la volée. »

**Trois avantages cumulés** Grace à l’efficacité de la reformulation de Python en MicroPython , nous cumulons trois avantages :

- La vaste communauté des programmeurs Python gagne l’opportunité de s’aventurer dans le développement de systèmes embarqués,
- Les développeurs de systèmes embarqués qui travaillent habituellement en langage C ont la possibilité de découvrir Python et la richesse de ses librairies,
- Les débutants en programmation ont rapidement accès à des fonctions très puissantes, qui permettent de développer rapidement et facilement des projets complets, mêlant contrôle, visuel, sonore, pilotage d’actionneurs, communication réseau, . . .

#### 1.6.6 Quelques cartes qui supportent le langage MicroPython

Cette liste est amenée à s’agrandir au fil des développements. Voir figure 2 page 7.

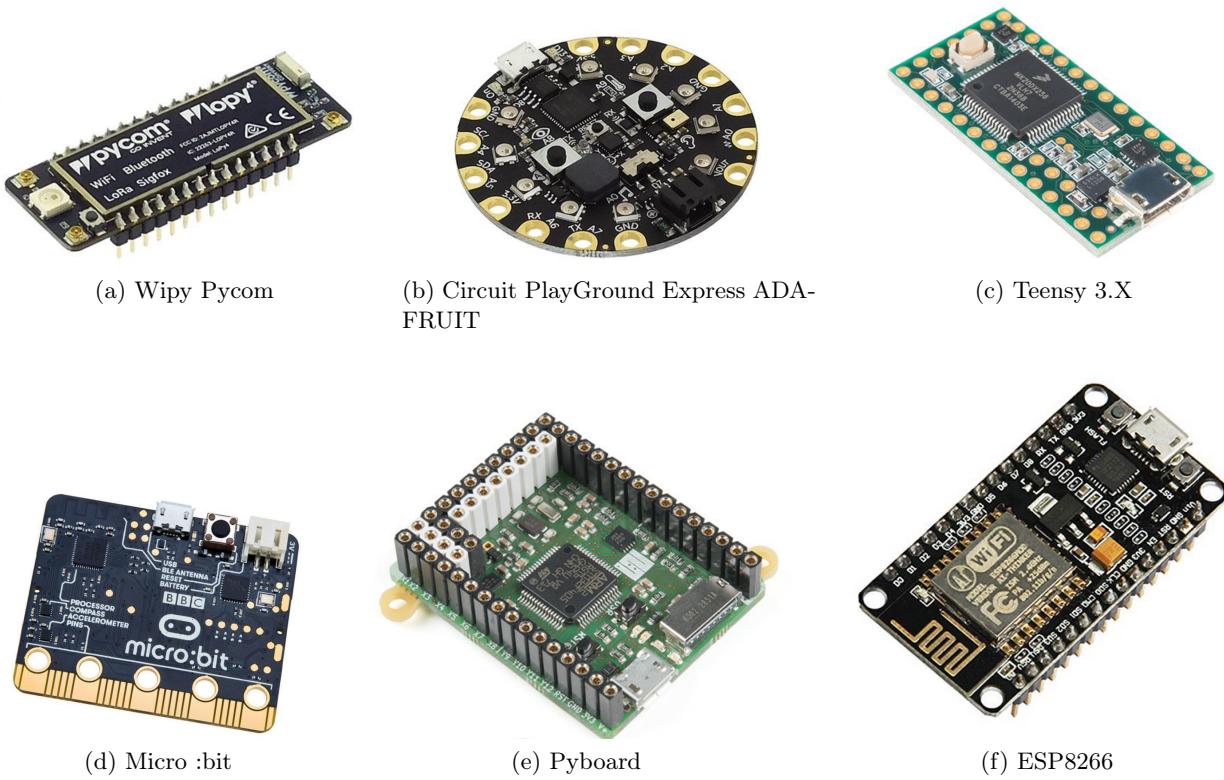


FIGURE 2 – cartes pour MicroPython

#### Liste des cartes qui supportent le langage MicroPython.

### 1.7 Pourquoi vouloir utiliser des appareils si petits et si peu puissants ?

#### 1.7.1 La notion de puissance est très relative

Certes les micro-contrôleurs sont peu puissants, mais seulement par rapport à un ordinateur de bureau, alors que ce dernier est prévu pour du calcul intensif bien souvent, pour faire tourner des jeux ultra rapides, ou de lourds logiciels de CAO, simulation numérique, photo, ou vidéo, . . .

Mais de quelle puissance de calcul a-t-on besoin pour piloter en vitesse par exemple un moteur ? Infiniment moins en réalité. Et pour allumer une lumière pendant 2 minutes suite à la présence d’une personne devant un capteur, ou pour faire défiler une barrière lumineuse ? Encore moins !

►Q.1: Est-il plus facile en terme de ressource processeur (la puissance de calcul) de piloter l'injection d'essence dans un moteur à combustion interne qui tourne à 6000 tours par minute (soit 100 tours par seconde), ou d'afficher une vidéo youtube ? Réponse : la puissance de calcul nécessaire est bien plus faible dans le premier cas.

### 1.7.2 Une intelligence embarquée pour une air de magie

Et d'ailleurs, utiliser un micro-contrôleur permet de faire des choses extraordinaires, par le simple fait de pouvoir embarquer un système de contrôle dans n'importe quel objet de la vie courante.

Le système de commande a longtemps été une carte électronique, mais aujourd'hui cette carte ressemble à un ordinateur, car elle embarque un processeur, de la mémoire, du wifi, et aujourd'hui cette carte est souvent un micro-contrôleurs.

Ces systèmes de commande permettent des interactions avec le monde extérieur, ils permettent de réguler, automatiser, créer des interactions avec le monde extérieur, se connecter sur les différents réseaux, et en quelque sorte enchanter le monde.

Bien souvent, toutes ces opérations se font sans intervention humaine, les systèmes étant même parfois alimentés par des panneaux solaires pour une autonomie totale. En tout cas, pour le novice, cet aspect de la commande ressemble à de la magie.

## 1.8 L'ESP32, un micro-contrôleur très performant

L'ESP32 est un micro-contrôleur cadencé à 240 MHz et exécutant un code écrit en langage MicroPython hébergé dans sa mémoire Flash. Voir figure 3 page 9.

L'interpréteur (le logiciel) MicroPython est implanté dans cette même mémoire Flash ; il pèse environ 1,4 Mo, soit moins de la moitié de la capacité totale (4 Mo).

Ce microcontrôleur dispose d'interfaces WiFi et Bluetooth idéales pour les objets connectés. Des connecteurs latéraux mâles permettent d'enficher le module sur une plaque de montage rapide (connexions rapides) (ou «breadboard»). L'interface sans fil Wifi permet la création de point d'accès sans fil, l'hébergement d'un serveur, la connexion à internet et le partage des données par exemple.

Le module se programme directement à partir de l'IDE Thonny par exemple, et nécessite simplement un cordon microUSB. Voici ces caractéristiques :

- Alimentation : 5 Vcc via micro-USB
- Alimentation : 3,3 Vcc via broches Vin
- Microprocesseur : Tensilica LX6 Dual-Core
- Architecture : 32 bits
- Fréquence d'horloge : 240 MHz
- Mémoire SRAM : 512 ko (pour stocker les variables)
- Mémoire Flash : 4 Mo (pour stocker nos programmes)
- 10 E/S digitales compatibles PWM (Modulation de largeur d'impulsions)
- Interfaces : Entrées capacitatives, DAC (digital Analog Converter), ADC (Analog Digital Converter), I2C (Inter Integrated Circuit), SPI (Serial Peripheral Interface), UART (Universal Asynchronous Receiver/Transmitter),
- Interface Wifi 802.11 b/g/n 2,4 GHz (jusqu'à 150 Mbits/s)
- Antenne Wifi intégrée

**Pin OUT de la carte ESP32 - 2** Voir figure 4 page 10.

Sur ce diagramme des broches (Pin-OUT), toutes les broches peuvent être des sorties ou des entrées numériques (sauf 34,35,36,39 qui ne peuvent être que des entrées) ; mais certaines broches peuvent aussi avoir d'autres fonctions. Par exemple, la plupart des broches peuvent être des entrées analogiques. Elles sont notées ADC, et sont au nombre de 16

## 2 Installations diverses

Ces installations ne seront à réaliser qu'une seule fois, lors de la première utilisation, et il sera nécessaire d'installer des drivers, utilitaires, etc ... Pour les séances de TP de l'ENIM, les installations sont déjà effectuée, vous pouvez sauter à la section

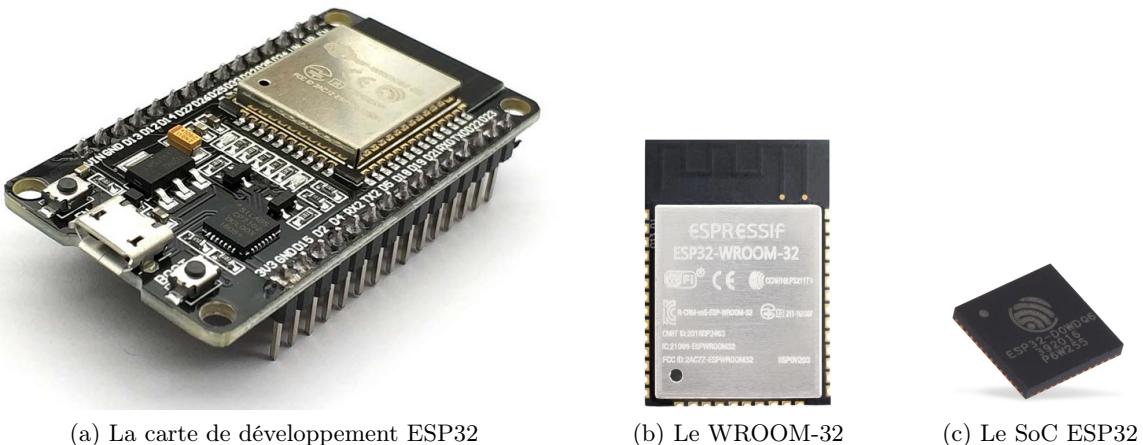


FIGURE 3 – ESP32

## 2.1 Installation de l'IDE Thonny

### 2.1.1 Définition de IDE ou «Integrated Development Environment»

On appelle IDE un environnement de développement intégré. C'est un ensemble d'outils informatique, conçu pour les programmeurs, destiné à assister et formaliser le travail de création de logiciels.

Un IDE comporte un «éditeur de texte» destiné à la programmation, ainsi que des fonctions qui permettent de démarrer le compilateur ou le «débogueur» qui permet d'exécuter ligne par ligne le programme en cours de construction.

Tous les outils de l'IDE sont prévus pour être utilisés ensemble. Cela permet d'augmenter la productivité des programmeurs en automatisant une partie des activités et en simplifiant les opérations.

Certains environnements sont dédiés à un langage de programmation en particulier ; c'est le cas de Thonny IDE, qui est à mon avis le mieux adapté pour une initiation à la programmation de l'ESP32.

### 2.1.2 L'IDE Thonny

Thonny est un IDE minimalistre qui permet d'apprendre le Python ; il a été mis au point par l'Université de Tartu en Estonie. Conçu pour les débutants, cet outil intègre son propre interpréteur Python 3.7 (mais vous pouvez aussi utiliser le vôtre), et offre des fonctionnalités plutôt pratiques quand on est dans un processus d'apprentissage. Il est intuitif, simple, en Open Source, et assez performant pour une prise en main. De plus, il est compatible avec Windows, MacOS, et Linux.

Nous utiliserons Thonny IDE, qui est normalement inclus sur notre distribution RaspBian du Raspberry. Pour MacOS et windows, il faudra l'installer. Lorsque nous installons Thonny IDE, il vient avec la dernière version de Python, soit la version 3.7 à ce jour.

L'avantage d'utiliser Thonny IDE est que non seulement il permet d'accéder à son propre interpréteur Python sur l'ordinateur, mais il peut aussi accéder à un interpréteur déporté dans une carte externe, par exemple celui qui sera téléchargé dans notre contrôleur ESP32. Ainsi, nous pourrons programmer en MicroPython notre carte ESP32.

### 2.1.3 Installation de l'IDE Thonny

*Attention :* Cette installation n'est à réaliser qu'une seule fois. Pour les séances de TP de l'ENIM, l'installation est déjà effectuée, vous pouvez sauter à la section 2.1.4.

**Sur Raspberry et Raspbian :** Normalement, pour Linux RaspBian, Python 3 et Thonny IDE sont installés ; sinon voici la procédure d'installation :

- Ouvrir le shell de commande de RaspBian, et suivre la procédure :
- Pour installer Python, saisir : `sudo apt install python3 python3-pip python3-tk`
- Pour installer Thonny IDE, saisir : `sudo apt install python3-thonny`
- Puis lancer Thonny, saisir : `thonny`

# ESP32 DEVKIT V1 – DOIT

version with 30 GPIOs

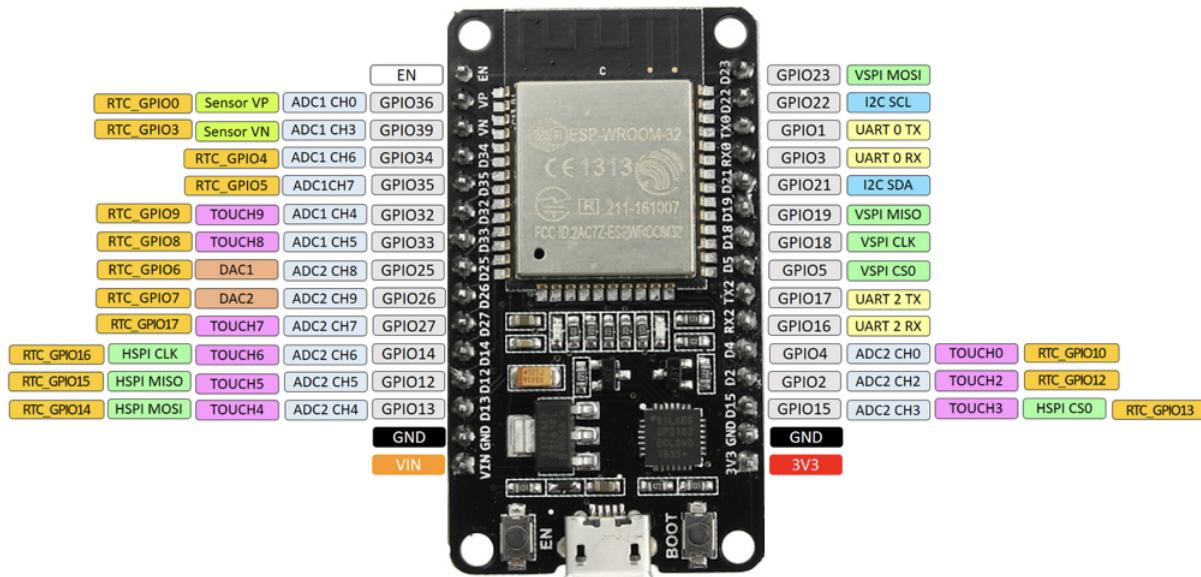


FIGURE 4 – PIN OUT de l'ESP32 30 broches

**Sur macOS et Windows :** L'installation se fait de manière classique, comme tout logiciel : [Lien vers le site officiel pour installer Thonny IDE.](#)

#### 2.1.4 Test de l'IDE Thonny

Thonny propose deux fenêtres, voir figure 6 :

- la console (ou le «shell»), en bas, pour saisir des instructions qui seront exécutées immédiatement, à chaque instruction saisie,
  - la fenêtre d'édition de code, en haut, pour saisir un script (un ensemble d'instructions) qui sera exécuté en une fois lors de la demande d'exécution.

►Q.2: Afficher la fenêtre des variables en allant dans l'onglet «View», et activer l'onglet «Variables».

Nous allons maintenant demander à Thonny d'utiliser Python 3.7 de l'ordinateur pour exécuter les instructions suivantes. Pour cela, allons dans le menu : «Tools», «Options», «Interpréteur», et choisir : «Le même interpréteur qui exécute Thonny (par défaut)». Voir figure 5 page 10.

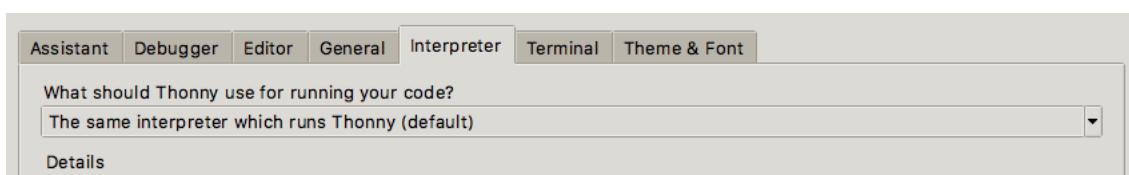


FIGURE 5 – Choix de l'interpréteur Python

►Q.3: Dans la console, saisir les trois lignes successivement, comme fait sur la figure 6, et remarquer l'apparition des variables, puis l'affichage du résultat.

►Q.4: Toujours dans la console (le shell), presser les flèches «flèche du haut» et «flèche du bas» pour vérifier que toutes les instructions saisies sont gardées en mémoire. S'en servir pour affecter maintenant

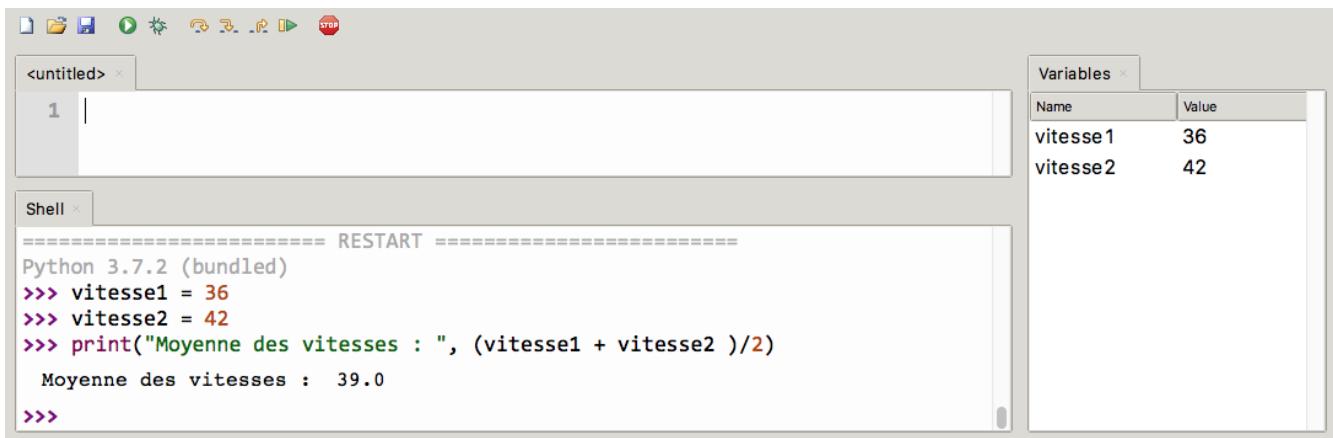


FIGURE 6 – L'IDE Thonny

la valeur 45 à ma variable `vitesse2`, puis afficher à nouveau la moyenne des vitesses.

### 2.1.5 Connection entre Thonny IDE et l'ESP32

Dans un second temps, ce sera la carte ESP32 qui exécutera ces instructions. Pour une première utilisation, notre carte ESP32 n'est pas encore «flashée» avec le firmware, c'est à dire avec le programme qui va interpréter le code MicroPython. Pour une première installation, il nous faut encore passer par les étapes suivantes :

- télécharger le firmware MicroPython, qui est l'interpréteur MicroPython implanté dans l'ESP32, et qui interprète une à une les instructions saisies dans la console, ou les lignes de codes du script,
- installer esptool.py à partir de Thonny,
- installer un driver si besoin pour dialoguer avec l'ESP32 (pour macOS et Windows),
- connecter la carte ESP32 à l'ordinateur,
- téléverser ce firmware dans la carte ESP32. On parle de «flasher» la carte. Il existe plusieurs outils pour faire cela. Les deux plus simples sont l'installation de l'utilitaire «esptool» pour Raspberry, et Thonny IDE pour macOS et Windows.
- indiquer à Thonny IDE que c'est le firmware qui doit exécuter les instructions.

*Attention :* Cette installation n'est à réaliser qu'une seule fois, lors de la première utilisation. Pour les élèves de l'ENIM, les cartes sont prêtes à l'emploi car le firmware est installé. Vous pouvez donc sauter à la section 2.2.4. Il sera nécessaire de faire toute cette la procédure lorsque vous utiliserez une carte ESP neuve pour un projet personnel ou autre.

## 2.2 Préparer le téléchargement du firmware dans l'ESP32 (Flasher le ESP32)

### 2.2.1 Télécharger le firmware MicroPython

**Téléchargement de la dernière version du FirmWare MicroPython pour ESP32.** Voir figure 7 page 12. Par exemple, j'ai téléchargé la version GENERIC-SPIRAM, c'est à dire qui fonctionne avec une mémoire pSRAM externe de 4 MO, celle qui correspond à notre ESP32 :

`esp32spiram-idf3-20191105-v1.11-555-g80df377e9.bin`

### 2.2.2 Installation de l'utilitaire «esptool.py»

Cet utilitaire est fourni par «Espressif», la société qui produit l'ESP32.

Esptool est codé en Python, il est open-source, indépendant de la plateforme, et permet de communiquer avec l'ESP32, pour lui téléverser l'interpréteur MicroPython que nous venons de télécharger.

**Sur Raspbian** Il est nécessaire d'avoir Python 2.7 ou 3.4 minimum installé sur votre ordinateur, ce qui est normalement déjà fait lorsque nous avons installé Thonny IDE.

L'instruction pour installer esptool est :

```
$ pip install esptool
```

En cas de difficulté d'installation, essayer l'une ou l'autre des solutions :

## Firmware for ESP32 boards

The following files are daily firmware for ESP32-based boards, with separate firmware for boards with and without external SPIRAM. Non-SPIRAM firmware will work on any board, whereas SPIRAM enabled firmware will only work on boards with 4MiB of external pSRAM.

Program your board using the esptool.py program, found [here](#). If you are putting MicroPython on your board for the first time then you should first erase the entire flash using:

```
esptool.py --chip esp32 --port /dev/ttyUSB0 erase_flash
```

From then on program the firmware starting at address 0x1000:

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20190125-v1.10.bin
```

Firmware built with ESP-IDF v3.x, with support for LAN and PPP but no bluetooth:

- o GENERIC : [esp32-idf3-20191105-v1.11-555-g80df377e9.bin](#)
- o GENERIC : [esp32-idf3-20190529-v1.11.bin](#)
- o GENERIC : [esp32-idf3-20190125-v1.10.bin](#)
- o GENERIC : [esp32-idf3-20180511-v1.9.4.bin](#)
- o GENERIC-SPIRAM : [esp32spiram-idf3-20191105-v1.11-555-g80df377e9.bin](#)
- o GENERIC-SPIRAM : [esp32spiram-idf3-20190529-v1.11.bin](#)

FIGURE 7 – Site de téléchargement du firmware pour l'ESP32

```
$ python -m pip install esptool  
$ pip2 install esptool
```

**Sur MacOS ou windows** Aller dans l'onglet de Thonny «Outils», puis «Gérer les Plugins», saisir dans «Rechercher un paquet sur Pypi» : esptool, et l'installer. Il est nécessaire de relancer Thonny IDE après cette installation.

### 2.2.3 Installation du driver CP2102

Ce driver permet de communiquer avec le circuit CP2102 de notre carte ESP32. Ce circuit adapte la liaison USB de l'ordinateur avec l'interface **UART** (Universal Asynchronous Receiver Transmitter, c'est un émetteur-récepteur asynchrone universel) de l'ESP32. Ce driver n'est pas installé par défaut sur MacOS ou windows, mais il l'est souvent sur les distributions Linux (Raspbian par exemple). Vous pouvez trouver ce driver avec Google en saisissant CP2102 driver download, ou ici :

[Téléchargement du driver CP2102.](#)

### 2.2.4 Connecter la carte au port UBS de l'ordinateur

Connecter à l'aide du câble micro-USB. Cette connexion alimente aussi la carte en 5V.

Aller dans l'onglet «Outils», «Options», puis «Interpréteur», et choisir «MicroPython (ESP32)». Ensuite, choisir le port sur lequel l'ESP32 est connecté à votre ordinateur. Voir figure 8.

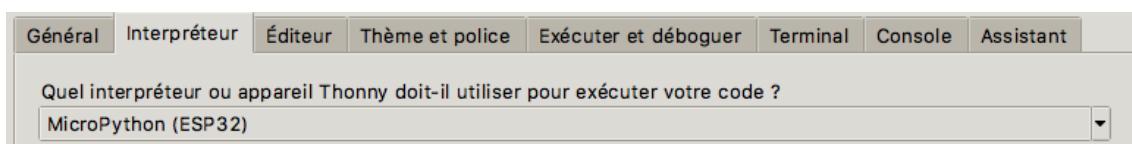


FIGURE 8 – Site de téléchargement du firmware pour l'ESP32

Si notre carte n'est pas encore chargée avec l'interpréteur MicroPython, ce qui est le cas pour une première utilisation, normalement Thonny répond :

Device is busy or does not respond. Your options :  
 - wait until it completes current work,  
 - use Ctrl+C to interrupt current work,  
 - use Stop/Restart to interrupt more and enter REPL.

... ce qui signifie que Thonny ne peut établir de communication avec la carte. Nous allons donc maintenant téléverser MicroPython dans la carte, ce que l'on appelle aussi «flasher» la carte. Ainsi, l'ordinateur et la carte pourront établir une communication ...

Pour les cartes utilisées en TP à l'ENIM, elles sont «flashées» avec un MicroPython récent. Pas besoin d'en installer un nouveau. Vous pouvez aller à la section 2.2.6.

### 2.2.5 Flasher notre carte ESP32

**Procédure sur MacOS et windows :** Dans la même fenêtre que précédemment, cliquer sous «Microlgiciel» sur «Ouvrir la boîte de dialogue pour installer ou mettre à jour MicroPython sur votre appareil». Voir figure 9.

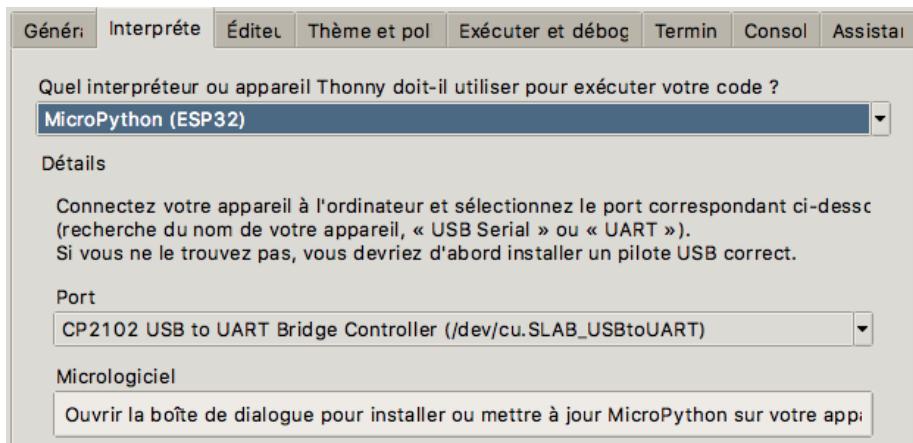


FIGURE 9 – Préparation au téléchargement du firmware dans l'ESP32

Ensuite, choisir le port de communication, parcourir les fichiers pour choisir votre firmware qui a l'extension «.bin». Et laisser le temps (1 minute environ) à Thonny pour effacer la carte, puis téléverser le firmware. Voir figure 10.

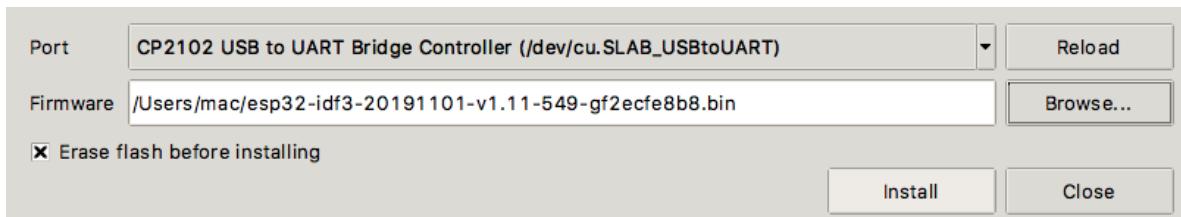


FIGURE 10 – Téléversement du firmware dans l'ESP32

**Procédure sur MacOS et windows :** Il nous faut effacer la carte, puis téléverser le nouveau FirmWare. Saisir ces instructions dans la console (ou le Shell) :

```
— esptool.py --chip esp32 --port /dev/cu.SLAB_USBtoUART erase_flash
— esptool.py --chip esp32 --port /dev/cu.SLAB_USBtoUART --baud 460800
  write_flash -z 0x1000 esp32-20191011-v1.11-422-g98c2eabaf.bin
```

Le port sur lequel est connectée votre carte est à modifier en fonction de votre ordinateur \*. Le fichier à téléverser est à modifier aussi; il doit être placé dans le répertoire de base de votre ordinateur.

\* : Normalement, il n'est pas nécessaire d'indiquer le port sur lequel votre carte est connectée, car «esptool.py» va énumérer tous les ports série connectés, et va essayer chacun d'eux pour trouver un composant «Espressif» connecté. Ainsi, on écrira simplement :

```
— esptool.py --chip esp32 erase_flash
```

```
— esptool.py --chip esp32 --baud 460800
write_flash -z 0x1000 esp32-20191011-v1.11-422-g98c2eabaf.bin
```

## 2.2.6 Fin de l'installation

A partir de maintenant, Thonny est paramétré pour envoyer les instructions de l'utilisateur vers MicroPython, qui exécutera les instructions. Nous allons pouvoir découvrir le monde merveilleux de Python embarqué sur une carte à microcontrôleur.

Parfois il est utile de cliquer sur le bouton STOP de Thonny, pour qu'il se connecte avec la carte. Une fois la carte connectée, voici le message que vous obtenez en figure 11, avec la version du firmware embarqué :

```
MicroPython v1.11-549-gf2ecfe8b8 on 2019-11-01; ESP32 module with ESP32
Type "help()" for more information.
>>>
```

FIGURE 11 – Connexion valide avec la carte via MicroPython

# Première partie

## Les bases

### 3 Introduction à Thonny et Python

#### 3.1 Prise en main de l'IDE Thonny et introduction à Python

##### 3.1.1 Utiliser Thonny comme un interpréteur

Thonny possède un espace dans lequel il est possible de saisir directement les commandes. Cette zone est appelée «console». Saisies dans la console, les instructions sont interprétées directement. Comme nous avons dit que l'interpréteur est MicroPython dans l'ESP32, chaque instruction provoque un accès à l'ESP32. A partir de cette console, nous allons découvrir les bases de Python ...

##### 3.1.2 Les opérateurs mathématiques

►Q.5: Saisir dans la console les commandes suivantes.

A chaque instruction saisie, il y a un transfert d'information. En effet, c'est MicroPython (implémenté sur l'ESP32) qui reçoit l'instruction à interpréter (transfert depuis l'ordinateur vers la carte), et qui renvoie le résultat de l'opération (un deuxième transfert, mais cette fois depuis la carte vers l'ordinateur).

```
>>> 2 + 3 * 4      # remarquez que la multiplication est prioritaire
14
>>> 15 / 3 - 1    # la division retourne un type "float"
4.0
>>> 13 // 4       # la division entière retourne un type "int"
3
>>> 13 % 2        # le modulo retourne le reste de la division entière
1
```

##### 3.1.3 Les «labels» ou «étiquettes» - Attacher des noms aux nombres

Pour concevoir des programmes qui vont au delà de la simple opération arithmétique, nous devons affecter des noms aux nombres. Par exemple :

```
>>> a = 4          # affectation du label a au nombre 4
>>> a + 1
5
```

Nous avons donné le nom **a** au chiffre 4. Ainsi, **a** renvoie au nombre 4, ce qui est pratique car il suffit de changer le nombre vers lequel **a** pointe et Python se sert de cette nouvelle valeur dans la suite du code. Ce type de nom est appelé «label», ou «étiquette». On parle alors d'assigner (ou affecter) le label **a** au nombre 4.

**Remarque concernant le terme «variable» :** On entend souvent le terme «variable» qui décrit la même chose que notre «label». Toutefois, étant donné que le terme «variable» est aussi un terme mathématique utilisé au *x* dans l'équation par exemple  $x - 1 = 1/x$ , nous l'utiliserons uniquement dans le cas d'équations ou expressions mathématiques.

##### 3.1.4 Types des données

Il existe 4 types de données :

Type de donnée	Description
int	Entier (positif ou négatif)
float	Flottant : nombre à virgule
str	String : chaîne de caractère
bool	Booléen : Vrai ou Faux

►Q.6: Saisir dans la console les commandes suivantes ; pour chaque donnée, on afficher le type de l'objet pour comprendre.

```
>>> x = 6          # assignation du label x au nombre 6
>>> type (x)      # retourne le type de la donnée
<class 'int'>    # retourne le résultat : classe entier
>>> y = x / 2     # retourne un float, même si x et 2 sont entiers
>>> y              # demande à afficher la valeur de y
3.0                # valeur affichée, et le point indique un type "float"
>>> type (y)
<class 'float'>
>>> message = " Bonjour à tous ! "   # identique à ' Bonjour à tous ! '
>>> type (message)
<class 'str'>
>>> e = True
>>> type (e)
<class 'bool'>
```

### 3.1.5 Changer les types de données

Nous pouvons changer les types de données, par exemple un type flottant en type entier, ou type chaîne de caractères. float -> int int -> float int -> str int -> bool (implicite)

```
>>> i = int(y)        # transformation en entier
>>> i
3
>>> type(i)
<class 'int'>
>>> f = float(2*i)    # transformation en flottant
>>> f
6.0
>>> texte = str(f)    # transformation en chaîne de caractère
>>> texte
'6.0'
>>> c = texte * 2
>>> c
'6.06.0'           # la chaîne est dupliquée
```

### 3.1.6 La fonction print

Elle affiche dans la console le message entre parenthèses. Cela sera pratique pour le débogage, durant l'exécution d'un script, que nous verrons plus loin.

```
>>> print("J'aime le soleil !")
J'aime le soleil !
```

### 3.1.7 Les opérateurs de comparaison

Opérateurs	Description
==	Egal à
!=	Non égal à
>	Supérieur à
<	Inférieur
>=	Supérieur ou égal à
<=	Inférieur ou égal à

►Q.7: Saisir dans la console les commandes suivantes ; bien comprendre que l'opérateur == teste l'égalité, ce qui n'est pas équivalent à l'opération d'affectation qui est =.

```
>>> 2 == 3
False
```

```
>>> x == 6      # test si x = 6 ! on n'affecte pas le label x au nombre 6
True
>>> 3 > 2
True
```

## 3.2 Utilisation de la zone d'édition des scripts

Jusqu'à maintenant, nous avons saisi des instructions dans la console, et elles étaient exécutées instantanément. Maintenant, nous allons saisir nos instructions dans la zone d'édition des scripts, qui permet d'exécuter plusieurs ligne de code à la suite, et de pouvoir faire des structures de boucle et de condition.

### 3.2.1 Une boucle simple avec la structure While

La structure `While` permet d'exécuter en boucle des instructions, avec une condition d'arrêt. Le code suivant affiche les nombres entre 1 à 4, c'est à dire que 4 boucles sont exécutées.

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

On obtient cela :

```
1
2
3
4
```

Bien remarquer l'indentation (retrait de 2 ou 4 caractères «<espace», ou TAB) pour les 2 lignes contenues dans le `While` (les lignes 3 et 4). C'est ce qui permet à Python de comprendre que ce sont ces deux lignes qui seront exécutées en boucle, tant que le label `nombre` est inférieur à 5.

►Q.8: Exécuter ce code, en cliquant sur la flèche verte (le raccourci est F5). Remarquer que pour une première exécution, Thonny demande à enregistrer ce code. Nous l'enregistrons sur l'ordinateur (pas sur la carte ESP32), et dans votre espace personnel, sous le nom : «Test1».

### 3.2.2 Une boucle simple avec la structure for

La boucle `for` permet de répéter une boucle un certain nombre de fois, pour par exemple parcourir les éléments d'une liste. Par exemple, la boucle ci dessous est exécutée pour toutes les valeurs de nombre comprise entre 1 et 4. La fonction `range()` crée une sorte de liste d'entiers, du premier (inclus) au dernier (exclus!). On obtient alors la même chose que précédemment.

```
for nombre in range(1, 5):
    print(nombre)
```

*Attention avant de saisir ce code!* : afin de ne pas créer un nouveau fichier à chaque exemple, nous allons mettre en commentaire les 4 premières lignes avec le symbole `#`, sauter une ligne, et saisir les suivantes. Voir ci-dessous. Cela permet d'avoir accès très rapidement à toutes les lignes déjà saisies pour pouvoir récupérer des bouts de code. Nous ferons de même pour tous les exemples qui suivent.

```
1  # nombre = 1
2  # while nombre < 5:
3  #     print(nombre)
4  #     nombre = nombre + 1
5
6  for nombre in range(1, 5):
7      print(nombre)
```

►Q.9: Saisir ce code.

Le fait que le deuxième nombre (5) soit exclus dans l'instruction `range(1, 5)` est assez logique : si nous ne mettons qu'une valeur, par exemple 5, la fonction `range(5)` affiche 5 valeurs, et comme la première est toujours 0, elle finit bien à 4.

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

►Q.10: Tester cela.

### 3.3 Utilisation des listes

#### 3.3.1 Création d'une liste

Une liste est une structure itérative, définie comme la ligne 1 ci dessous par exemple. Elle est itérative car on peut parcourir ses éléments un à un à l'aide d'une boucle `for`.

La structure «liste» est très souvent utilisées dans la programmation Python ; nous allons donc y passer quelques instants pour bien la comprendre.

Par défaut, nous avons un retour à la ligne après chaque `print()` ; le paramètre `end=" "` dans la fonction `print()` permet de changer le caractère de fin de l'affichage, et de remplacer donc le retour à la ligne par un espace. Cela permet d'afficher tous les éléments de la liste sur la même ligne. Dans l'exemple suivant, on affiche *un par un* les éléments de la liste qui contient tous les carrés des nombres entre 1 et 10.

```
1 ma_liste = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
2 for element in ma_liste :
3     print(element, end=" ")
```

Pour obtenir cela :

```
1 4 9 16 25 36 49 64 81 100
```

►Q.11: Tester le bon fonctionnement.

Pour vérifier que les éléments sont affichés un par un, on peut mettre un délai (0.5 s par exemple) entre deux itérations pour voir défiler les 10 valeurs. Pour cela, nous avons ajouté la méthode `sleep` (nous verrons plus tard ce que cela veut dire) du module `time`, qui permet d'attendre un temps indiqué en secondes.

```
1 from time import sleep
2 ma_liste = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
3 for element in ma_liste :
4     print(element, end=" ")
5     sleep(0.5)
```

Si nous voulions juste connaître la valeur de `ma-liste`, on aurait pu écrire (dans la console par exemple, ou à la place des lignes 3 à 5) directement et plus simplement, en une seule opération d'affichage, l'instruction suivante :

►Q.12: Saisir dans la console l'instruction suivante :

```
1 >>> print(ma_liste)
2 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Dans ce cas, `ma-liste` est affichée entre crochets (comme une liste). Attention, on ne réalise pas la même opération. Précédemment, on prenait un à un chaque élément de la liste pour l'afficher, ou en faire un traitement.

►Q.13: Vérifier que notre variable figure bien dans la fenêtre des variables.

#### 3.3.2 Création d'une liste avec la méthode «append»

La méthode «append» s'applique à une liste, et permet d'ajouter un élément à cette liste, à la fin, qui est ainsi modifiée. En utilisant cette méthode pour créer et afficher la même liste, nous pouvons itérer de 1 à 11, et ajouter à `ma_liste_calc` («calc» pour calculée) un élément à chaque itération. On commence par créer une liste vide en ligne 1.

L'instruction `**` correspond à l'opération mathématique puissance.

►Q.14: Tester le bon fonctionnement, c'est à dire afficher tous les éléments de `ma_liste_calc`. Il faudra ajouter une ligne ou deux à ce code.

```
1 ma_liste_calc = []
2 for x in range(1, 11):
3     ma_liste_calc.append(x**2)
```

### 3.4 Comment tester si un nombre est pair ou multiple de $x$ ?

Il est très fréquent en programmation d'avoir à tester si un nombre est multiple d'une valeur  $x$ . Nous allons nous en servir très souvent dans cet ouvrage.

**Principe de la division et de son reste.** Pour qu'un nombre soit pair, le principe est de le diviser par 2, et de tester si le reste de la division vaut 0 ; s'il vaut 0, le nombre est pair. Exemple :

- $13/2 = 6,5$  ; la valeur entière de la division vaut 6 et le reste vaut 1 car  $13 = 6 * 2 + 1$ . Donc le nombre 13 n'est pas pair.
- $12/2 = 6$  ; la valeur entière de la division vaut 6 et le reste vaut 0 car  $12 = 6 * 2 + 0$ . Donc le nombre 12 est pair.

Le principe est le même pour savoir s'il est multiple de 5 : on divise par 5 au lieu de 2. Exemple :

- $13/5 = 2,6$  ; la valeur entière de la division vaut 2 et le reste vaut 3 car  $13 = 2 * 5 + 3$ . Donc le nombre n'est pas multiple de 5.
- $15/5 = 3$  ; la valeur entière vaut 3 et le reste vaut 0 car  $15 = 3 * 5 + 0$ . Donc le nombre est multiple de 5.

**Utilisation de l'instruction «modulo» %.** Pour faire le calcul du reste de la division, il est courant d'utiliser une opération plus adaptée, qui est l'instruction «modulo», qui s'écrit `%`, et qui retourne directement le reste de la division, car le calcul de la partie entière ne nous avance pas pour tester si un nombre est divisible par un autre. Exemple, on teste si  $n$  est multiple de 5, pour les valeurs  $n = 13$  et  $n = 15$  :

```
1 n = 13
2 print(n, "% 5 = ", n % 5)
3 print(n,"multiple de 5 ? : ", (n % 5) == 0)
4 n = 15
5 print(n, "% 5 = ", n % 5)
6 print(n,"multiple de 5 ? : ", (n % 5) == 0)
```

Ce qui nous donne :

```
13 % 5 = 3
13 multiple de 5 ? : False
15 % 5 = 0
15 multiple de 5 ? : True
```

►Q.15: Tester le bon fonctionnement.

►Q.16: Ajouter le code pour tester si 27 est multiple de 5.

## 3.5 Procédures et fonctions

### 3.5.1 Crédation de la procédure «multiple»

Une procédure est une série d'instruction :

- qui porte un nom,
- qui effectue un travail dont le résultat dépend de variables appelées arguments,
- et qui ne renvoie pas de résultat.

Pour définir une procédure, on utilise la commande `def`.

Pour simplifier le code, et ne pas réécrire les lignes 2 et 3, nous allons créer une procédure qui prend en argument  $n$ , et qui affiche les deux mêmes lignes que précédemment.

```
1 def multiple(n):
2     print(n, "% 5 = ", n % 5)
3     print(n,"multiple de 5 ? : ", (n % 5) == 0)
4
5 multiple(7)
6 multiple(10)
```

►Q.17: Ajouter le code pour tester si 27 est multiple de 5.

**Variante 1 :** On aurait pu aussi, au lieu d'afficher les deux lignes avec `print`, retourner seulement l'information (True ou False) sur le nombre, à savoir s'il est multiple de 5 ou pas. Dans faire cela, nous pouvons créer une fonction. Voyons cela ...

### 3.5.2 Cr ation de la fonction «multiple»

Une fonction est une s rie d'instruction :

- qui porte un nom,
- qui effectue un travail dont le r sultat d pend de variables appel es arguments,
- et qui renvoie une valeur que l'on peut stocker, si on le souhaite, dans une variable.

Pour d finir une fonction, on utilise la commande `def`, et on renvoie le r sultat (une valeur) gr ce   la commande `return`. Par exemple :

```
def multiple(n):  
    return (n % 5) == 0 # Retourne VRAI si n est multiple de 5  
  
print("Le chiffre 10 est multiple de 5 ? : ", multiple(10))
```

**Variante 2 :** On aurait pu aussi, au lieu de tester par rapport   5, tester par rapport   n'importe quel nombre,  $x$  par exemple, comme cela :

```
def multiple(n, x):  
    return (n % x) == 0 # Retourne VRAI si n est multiple de x  
  
print("Le chiffre 10 est multiple de 4 ? : ", multiple(10, 4))
```

**Variante 3 :** On va maintenant demander   l'utilisateur d'indiquer deux nombres entiers,  $n$  et  $x$ . La fonction `input` retourne une cha ne de caract re, que l'on transforme en nombre entier. On indiquera le r sultat sous la forme :

- « Le nombre 12 n'est pas multiple de 5, car la division de 12 par 5 donne 2, et le reste vaut 2.», ou
- « Le nombre 12 est multiple de 4, car la division de 12 par 4 donne 3, et le reste est nul.».

1 – `test-multiple.py` - Test si  $n$  est multiple de  $x$

```
1 def multiple(n, x):  
2     return (n % x) == 0  
3  
4 nombre = int(input("Quel est le nombre n a tester ? : "))  
5 diviseur = int(input("Quel est le diviseur a tester ? : "))  
6  
7 if multiple(nombre, diviseur) :  
8     print("Le nombre {0} est multiple de {1}, car la division de {0} par {1}  
      donne {2}, et le reste est nul.".format(nombre, diviseur, int(nombre/  
      diviseur)))  
9 else :  
10    print("Le nombre {0} n'est pas multiple de {1} car la division de {0}  
      par {1} donne {2}, et le reste vaut {3}. ".format(nombre, diviseur, int(  
      nombre/diviseur), nombre%diviseur))
```

►Q.18: Tester ce programme avec les valeurs 12 et 5 par exemple, puis 12 et 4.

►Q.19: Remarquer l'indentation apr s la structure `if`.

►Q.20: Tester ce programme avec un nombre non entier, avec 12.2 et 4 par exemple. Constater que cela g n re une erreur, et comprendre que la conversion en entier ne peut se faire que si la cha ne de caract re contient uniquement un nombre entier.

```
ValueError: invalid syntax for integer with base 10
```

### 3.6 Gestion des erreurs

Pour prendre en compte le fait que l'utilisateur pourrait entrer une valeur non conforme, nous pouvons utiliser la structure `try ... except`. S'il n'y a pas d'erreur lors de l'ex cution, la section `try` (essaie de faire ceci ...) se d roule normalement. En cas d'erreur, le programme saute   la section `except` (sinon faire cela ...).

## 2 – test-multiple-gestion-erreur.py - Test si n est multiple de x, et gestion erreur

```
1 def multiple(n, x):
2     return (n % x) == 0
3
4 try :
5     nombre = int(input("Quel est le nombre n à tester ? : "))
6     diviseur = int(input("Quel est le diviseur à tester ? : "))
7
8     if multiple(nombre, diviseur) :
9         print("Le nombre {} est multiple de {}".format(nombre, diviseur))
10    else :
11        print("Le nombre {} n'est pas multiple de {}".format(nombre, diviseur))
12
13 except :
14     print("Veuillez saisir des valeurs entières !")
```

►Q.21: Tester ce code en saisissant la valeur 12.2 et 4, puis 12 et 4.

## 3.7 Revenons sur les listes

### 3.7.1 Crédation d'une liste, et utilisation de la structure condition : «if»

Nous voulons ajouter une condition : nous créons la liste seulement pour les valeurs de  $x$  de 1 à 10 qui ne sont pas multiples de 5. Voici :

3 – liste-append.py - Crédation d'une liste par ajout d'éléments avec la méthode «append»

```
1 ma_liste = []
2 for x in range(1,11) :
3     if (x % 5) != 0 :          # si x n'est pas multiple de 5 :
4         ma_liste.append(x**2)  # ajout de x à la liste en construction
5 for element in ma_liste :
6     print(element , end="-")
```

On obtient cela :

```
1-4-9-16-36-49-64-81-
```

►Q.22: Tester le bon fonctionnement.

Nous vérifions qu'il manque bien le 25 et le 100.

### 3.7.2 Crédation d'une liste «en compréhension»

La crédation d'une liste en compréhension permet de créer une liste en une seule instruction, même s'il y a une condition à respecter. Dans l'exemple ci-dessous, nous créons la liste des carrés de tous les éléments entre 1 et 10, sauf les éléments multiples de 5. Ainsi, au lieu d'écrire les lignes 1 à 4 de l'exemple précédent, nous écrivons :

```
ma_liste = [x*x for x in range(1, 11) if (x % 5) != 0]
```

►Q.23: Vérifier que cela fonctionne bien en affichant la liste.

►Q.24: Créer et afficher la liste des nombres qui sont multiples de 7 OU de 11, et ceci entre 1 et 50. Afin de mieux comprendre, votre liste contiendra entre autres les valeurs 21 (car  $12 = 7 * 3$ ) et 22 (car  $22 = 11 * 2$ ).

Dans l'exemple suivant, nous faisons l'intersection de deux listes, c'est à dire la liste des éléments communs :

```
liste1 = [1, 3, 5, 7, 9]
liste2 = [5, 6, 7, 8, 9]
liste_elements_comuns = [value for value in liste1 if value in liste2]
print(liste_elements_comuns)
```

►Q.25: Comprendre le programme.

### 3.7.3 La fonction «enumerate»

Cette fonction permet, à partir d'une liste ou toute autre structure sur laquelle il est possible d'itérer, de fournir une valeur qui s'incrémente à chaque élément de la liste, et de l'associer à cet l'élément. Nous pouvons alors, avec une écriture simplifiée, afficher les éléments de la liste et leur numéro grâce à une boucle `for` d'une syntaxe un peu particulière : `element` prend successivement pour valeur chaque élément de la liste, pendant que `compt` s'incrémente en partant de 0...

```
1 ma_liste = [x*x for x in range(1,11) if(x % 5) != 0]
2 for compt, element in enumerate(ma_liste) :
3     print (compt, element)
```

►Q.26: Vérifier le bon fonctionnement.

Réponse à la question de la section 3.7.2 :

```
ma_liste = [x for x in range(1, 50) if (x % 7) == 0 or (x % 11) == 0 ]
```

## 3.8 Exemple de synthèse : PGCD

Le programme suivant fait la synthèse de tout ce que nous venons de voir. Ce programme calcule tous les diviseurs de deux nombres, puis tous les diviseurs communs, puis le plus grand d'entre eux, appelé le PGCD (Plus Grand Commun Diviseur). Ainsi, il va fournir un résultat sous la forme (exemple avec 20 et 35) :

```
x = 20 ; y = 70
Liste des diviseurs de x : [1, 2, 4, 5, 10, 20]
Liste des diviseurs de y : [1, 2, 5, 7, 10, 14, 35, 70]
Liste des diviseurs communs : [1, 2, 5, 10]
PGCD : 10
```

Pour comprendre le code suivant, et en particulier pour savoir si un nombre  $x$  est divisible par un nombre  $e$ , il suffit de tester si  $x/e$  donne un nombre entier, ou encore, nous l'avons déjà vu, si  $x \% e == 0$ .

4 – pgcd.py - Exemple du calcul du PGCD de deux nombres

```
1 # PGCD : programme de calcul du PGCD de 2 entiers
2 # Auteur : Vincent HERMITANT - ENIM 2020
3 x = 20
4 y = 70
5 diviseurs_x=[]      # creation liste vide pour diviseurs de x
6 diviseurs_y=[]      # creation liste vide pour diviseurs de y
7
8 # creation d'une fonction pour trouver les elements communs a deux listes
9 def inter(liste1, liste2):
10    liste_elements_comuns = [value for value in liste1 if value in liste2]
11    return liste_elements_comuns
12
13 for a in range (1,max(x,y)+1):
14    if x % a == 0 :          # si x est divisible par a
15        diviseurs_x.append(a) # ajout a la liste des diviseurs de x
16    if y % a == 0 :          # si y est divisible par a
17        diviseurs_y.append(a) # ajout a la liste des diviseurs de y
18 diviseurs_comuns = inter(diviseurs_x, diviseurs_y)
19 plus_grand_commun_diviseur = max(diviseurs_comuns) # retourne le maxi
20
21 print("x = ", x, "; y = ", y)
22 print ("Liste des diviseurs de x :",diviseurs_x)
23 print("Liste des diviseurs de y :",diviseurs_y)
24 print("Liste des diviseurs communs : ", diviseurs_comuns)
25 print("PGCD : ", plus_grand_commun_diviseur)
```

►Q.27: Comprendre ce code.

## 4 Considérations sur la programmation orientée objet et les modules

### 4.1 Découverte de la programmation orientée objet

#### 4.1.1 Introduction - Définitions

Python est un langage de programmation orienté objet (OOP : Objet-Oriented Programming). En python, tout est objet. Quand vous croyez utiliser un nombre, une variable, une fonction, ce sont des objets qui se cachent derrière. Par exemple, quand nous avons écrit `ma_liste.append()`, nous avons appliqué la méthode `append` sur l'objet `ma_liste` qui est une liste d'entiers.

Il y a deux concepts importants à comprendre à propos de l'OOP : les «classes» et les «objets».

**Objet, méthodes, attributs :** Un objet est une sorte de variable, qui peut contenir elle-même des fonctions et des variables.

- Les fonctions contenues dans les objets sont appelées des «méthodes»,
- Les variables contenues dans les objets sont appelées des «attributs».

**Classe :** Une «classe» est un «prototype» de l'objet, une sorte de modèle à partir duquel on va construire un objet. C'est à l'intérieur de la classe que nous allons définir les attributs (variables contenues dans l'objet) et les méthodes (les fonctions) qui caractérisent l'objet. C'est une collection d'attributs (données, variables) et méthodes à l'intérieur d'une entité unique.

**Instanciation :** A partir d'une classe, on peut créer un objet, qu'on appelle une «instance de classe». A travers l'objet créé, nous pouvons utiliser toutes les fonctionnalités de sa classe.

Ok, tout cela a l'air bien compliqué. Prenons un exemple très simple pour comprendre le concept.

#### 4.1.2 Première approche - création directe de la classe

Nous voulons définir, par exemple, plusieurs personnes, dans un programme Python, en utilisant les mêmes attributs (variables). Les attributs seront, par exemple, son nom, son prénom, son âge, son lieu de résidence. Pour faire cela, nous pouvons utiliser l'entité «Personne», qui sera appelé une classe.

Créons alors cette classe appelée «Personne», qui aura les attributs : nom, prénom, âge, lieu de résidence. Je choisis de ne rien renseigner à ce stade de l'exemple.

#### 5 – classe-1.py - Première approche des classes

```
1 class Personne :  
2     nom = ""  
3     prenom = ""  
4     age = 0  
5     lieu_residence = ""  
6  
7 personne1 = Personne()  
8 personne2 = Personne()  
9  
10 personne1.nom = "DELUNE"  
11 personne1.prenom = "Claire"  
12 personne1.age = "30"  
13 personne1.lieu_residence = "METZ"  
14  
15 personne2.nom = "CELERT"  
16 personne2.prenom = "Jacques"  
17 personne2.age = "32"  
18 personne2.lieu_residence = "NANCY"  
19  
20 print(personne1.nom)
```

- Lignes 1 à 5 : Nous définissons une classe, qui contient 4 variables (rappel : variable = attribut), qui ne possède aucune valeur valeur pour le moment, à part des 0 et des chaînes de caractères vides par exemple.

- Lignes 7 et 8 : Nous définissons autant de personnes que nous voulons en utilisant la classe. Nous créons les objets `personne1` et `personne2`, qui sont deux instances de la classe `Personne()`,
  - Lignes 10 à 18 : Pour chaque personne (c'est-à-dire chaque objet, ou encore pour chaque instance de la classe `Personne`), nous affectons le nom, prénom, âge, et lieu de résidence. Nous appelons cela «initialiser» chaque «attribut» de l'objet.
  - Ligne 20 : Nous affichons l'attribut `nom` de la personne 1.
- Q.28: Tester ce code, puis afficher l'âge de la personne 2.

#### 4.1.3 Deuxième approche - Utilisation d'un constructeur

Maintenant, nous allons utiliser un «constructeur», dont le rôle est de construire l'objet.

Le constructeur est une méthode spéciale de notre objet; c'est la méthode qui se charge de créer ses attributs, et qui est toujours appelée lors de la création de l'objet (rappel : objet = instance de la classe). Il est défini sur les lignes 2 à 6.

##### 6 – classe-2.py - Deuxième approche des classes

```

1 class Personne :
2     def __init__(self) :
3         self.nom = ""
4         self.prenom = ""
5         self.age = "0"
6         self.lieu_residence = ""
7
8 personne1 = Personne()
9 personne1.nom = "DELUNE"
10 personne1.prenom = "Claire"
11 personne1.age = "30"
12 personne1.lieu_residence = "METZ"
13
14 print(personne1.age)

```

Ici, le constructeur est vide, il est très courant pour les programmeurs de faire de cette manière lors de la création d'une classe. Vous ne verrez pas tout de suite l'intérêt, mais attendez de voir les deux approches suivantes pour poser la question.

- Q.29: Tester ce code.

Explication : Quand on tape `Personne()`, on appelle le constructeur de notre classe `Personne`, et celui-ci prend en paramètre une variable un peu mystérieuse : `self`. En fait, il s'agit tout simplement de notre objet en train de se créer. Il ne reste plus qu'à renseigner tous ses attributs un par un.

Mais ici encore ce n'est pas une manière de faire très élégante ni pratique.

#### 4.1.4 Troisième approche - Définition des attributs à la création de l'objet

Nous allons alors définir directement les attributs lors de la création de la classe, ainsi l'objet est déjà défini dès sa création par tous ses attributs. Voici le code :

##### 7 – classe-3.py - Troisième approche des classes

```

1 class Personne :
2     def __init__(self) :
3         self.nom = "DELUNE"
4         self.prenom = "Claire"
5         self.age = "30"
6         self.lieu_residence = "METZ"
7
8 personne1 = Personne()
9 print(personne1.age)

```

- Q.30: Tester ce code.

Tout cela fonctionne très bien. Pourtant, il y a un petit problème : avec cette manière de faire, tous les objets (les personnes) que nous allons créer possèdent par défaut les mêmes attributs, ce qui n'est encore une fois pas très élégant, même si nous pouvons les changer par la suite, comme déjà fait précédemment.

#### 4.1.5 Quatrième approche - Les paramètres sont passés lors de la création de l'objet

Pour faire un constructeur un peu plus intelligent, nous allons faire en sorte qu'il puisse prendre en paramètre les attributs. Voyons cela :

8 – **classe-4.py** - Quatrième approche des classes

```
1 class Personne :  
2     def __init__(self, nom, prenom, age, lieu_residence) :  
3         self.nom = nom  
4         self.prenom = prenom  
5         self.age = age  
6         self.lieu_residence = lieu_residence  
7  
8     personne1 = Personne("DELUNE", "Claire", 30, "METZ")  
9     personne2 = Personne("CELERT", "Jacques", 32, "NANCY")  
10    print(personne1.age)  
11    print(personne2.lieu_residence)
```

►Q.31: Tester ce code.

Cette fois, tout à l'air bien cohérent. Pour créer un objet (une personne), nous lui passons lors de l'instanciation tous les paramètres.

#### 4.1.6 Cinquième approche - Ajout à l'objet d'une méthode

Maintenant, nous allons créer la méthode (rappel : méthode = fonction) `description()`, qui va afficher toutes les informations sur une personne.

9 – **classe-5.py** - Cinquième approche des classes

```
1 class Personne :  
2     def __init__(self, nom, prenom, age, lieu_residence) :  
3         self.nom = nom  
4         self.prenom = prenom  
5         self.age = age  
6         self.lieu_residence = lieu_residence  
7     def description(self) :  
8         print("{1} {0} a {2} ans et habite à {3}.".format(self.nom,  
9                 self.prenom, self.age, self.lieu_residence))  
10  
11    personne1 = Personne("DELUNE", "Claire", 30, "METZ")  
12    personne1.description()
```

►Q.32: Tester ce code.

►Q.33: Ajouter un attribut à la classe, l'attribut «animal», et nous définirons que Claire a un chat.

►Q.34: Modifier la méthode `description` pour afficher à la suite de la phrase : « ... et possède un chat».

### Les classes, en résumé

Une classe est composée d'un constructeur qui initialise les attributs, et de l'ensembles des méthodes liées à cette classe. A partir de la classe, on crée les objets, appelés instances de classe.

## 4.2 Les modules et le module «machine»

### 4.2.1 Définition d'un module

Un module est en quelque sorte un bout de code enfermé dans un fichier, qu'on peut aussi appeler bibliothèque. Un module est composé de classes en général, mais aussi de fonctions et de variables qui ont un rapport entre elles. Nous pouvons utiliser nos propres modules, ou utiliser ceux fournis avec la distribution standard de Python.

Si l'on veut travailler avec les fonctionnalités prévues par le module, il suffit d'importer le module.

MicroPython est fourni avec seulement une petite partie des modules de Python car il doit pouvoir être implanté sur la carte ESP32, c'est-à-dire avec des contraintes matérielles particulières, qui n'existent pas sur un ordinateur classique, à savoir une taille mémoire très réduite en particulier. Par exemple, il n'a pas besoin des fonctions dédiées à l'affichage sur un moniteur car il n'en est pas pourvu.

En revanche, il a besoin de fonctions qui permettent d'accéder aux broches d'entrées/sorties (ou «pin», ou «pattes») du micro-contrôleur et au fonctionnalités matérielles. Le «module» `machine` est un ensemble de classes et fonctions dédiées à cela.

#### 4.2.2 Exemple avec le module `math` - bibliothèque standard de Python

Ce premier exemple utilise un module de la bibliothèque standard de Python. Par exemple, si vous tapez dans la console l'instruction suivante, elle provoque une erreur. Testez cette instruction :

```
>>> sin( pi/2 )
```

Mais si vous importez le module `math`, vous pouvez écrire :

```
>>> import math  
>>> math.sin( math.pi/2 )
```

Ou alors, si vous importez explicitement les deux fonctions `sin` et `pi`, on pourra écrire plus simplement :

```
>>> from math import sin, pi  
>>> sin( pi/2 )
```

#### 4.2.3 Exemple avec le module `esp` ou `esp32` - bibliothèque spécifique à l'ESP32

```
import esp  
esp.flash_size()      # retourne la taille de la mémoire Flash
```

#### 4.2.4 Le module `machine`

L'instruction suivante permet de rendre le `machine` module accessible à l'interpréteur MicroPython :

```
>>> import machine
```

►Q.35: Depuis la console, saisir cette instruction.

Nous pouvons par exemple connaître la fréquence de l'horloge du ESP32.

```
>>> machine.freq()          # retourne la fréquence de l'horloge de notre CPU
```

L'instruction ci-dessous rend accessible seulement la classe `Pin` du module `Machine` :

```
>>> from machine import Pin      # attention au P : MAJUSCULE
```

A partir de maintenant, nous pouvons créer les objets qui permettront d'accéder aux broches du micro-contrôleur et au fonctionnalités matérielles.

## 5 Généralités, et premiers tests sur les entrées et sorties numériques

### 5.1 Utilisation de la console pour lire ou écrire l'état d'une sortie numérique

Dans toute cette section, nos instructions seront saisies dans la console, une à une, pour une exécution et un retour d'information immédiat.

#### 5.1.1 Déclarer un broche (une pin) comme étant une sortie numérique

A partir de maintenant, la classe `Pin` est importée, et nous allons créer les objets qui permettront de lire une entrée numérique, ou d'écrire un état logique sur une sortie numérique.

►Q.36: Depuis la console, écrire l'instruction suivante qui permet de créer l'objet `led`, et de déclarer que la broche (ou pin) 2 est en sortie.

```
>>> led = Pin(2, Pin.OUT)
```

### 5.1.2 Ecrire un état sur une sortie numérique

Nous pouvons maintenant faire passer au niveau Haut ou Bas la sortie, et connecter une LED sur cette sortie pour qu'elle s'allume à l'état Haut.

Une LED bleue est intégrée sur le circuit ESP32, sur la pin 2. La broche 2 est la seule à être munie d'une LED.

De plus, une LED jaune a été câblée sur la platine de connexions, sur la sortie 18, selon le schéma de la figure 12. Nous remarquons la mise en série d'une résistance de limitation de courant de  $560\ \Omega$ , qui permet de régler un courant à la valeur d'environ  $2\ mA$  dans la LED ( $(3,3V - 2,2V)/560\Omega = 2mA$ ) lorsque la sortie est à l'état haut, soit  $3,3V$ . Rappel :  $2,2V$  est environ la tension de seuil de la LED, c'est à dire la tension à ses bornes lorsque un courant la traverse.

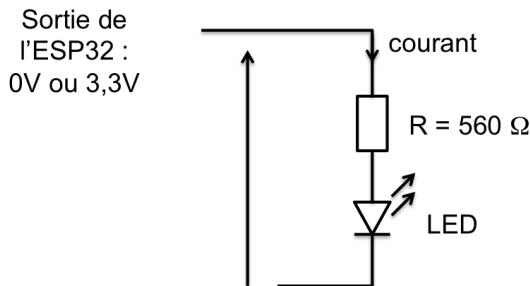


FIGURE 12 – Cablage de la LED, un niveau  $3,3V$  provoque son allumage

►Q.37: Maintenant, mettre la sortie (2) à l'état haut, et voir la LED bleue s'allumer.

```
>>> led.value(1)
```

Puis on l'éteint :

```
>>> led.value(0)
```

Ce que nous avons fait, c'est utiliser la méthode `value()` de l'objet `led`, pour changer l'état de la pin.

►Q.38: Tester aussi les instructions, et vérifier que c'est juste une autre écriture pour ces fonctions :

```
>>> led.on()
>>> led.off()
```

On aurait pu simplifier ; au lieu de créer l'objet `led`, puis de mettre sa valeur à 1, une seule ligne permet de faire les deux opérations en même temps. Cependant, cela est moins pratique pour la suite, par exemple lorsqu'on doit faire changer l'état de la Pin plusieurs fois.

```
>>> Pin(2, Pin.OUT).value(1)
>>> Pin(2, Pin.OUT).off()
```

Nous aurions pu faire changer d'état n'importe quelle broche de l'ESP32, car elles sont toutes disponibles en sortie numérique.

►Q.39: Faire changer l'état de la LED jaune connectée sur la sortie 18.

### 5.1.3 Utilisation d'une LED au format «Grove»

Pour découvrir un connecteur intéressant, nous allons ajouter une LED rouge, une nouvelle LED qui sera au format Grove, que nous allons câbler sur la sortie 19.

►Q.40: Connecter la LED Grove sur la sortie 19, et vérifier qu'il est possible de l'allumer et l'éteindre. Les fils sont repérés comme ceci :

- Noir : Masse.
- Rouge :  $3,3V$ .
- Blanc : fil non connecté.
- Jaune : tension qui passe de 0 à  $3,3V$ .

### 5.1.4 Lire un état sur une entrée numérique

Lire un état logique sur une entrée numérique, c'est lire une tension qui présente deux valeurs différentes, appelés niveaux. Cette tension doit varier suffisamment lors de l'appui par exemple sur un bouton poussoir, de manière à être détectable.

Dans l'idéal, cette tension devrait passer de 0V à 3,3V lors de l'appui (ou l'inverse), dans le cas d'une tension d'alimentation de 3,3V. En pratique, l'entrée reconnaît sans erreur le niveau logique dont la tension se situe dans deux plages de valeurs. Ainsi :

- si nous avons entre 2.5V et 3.6V, l'entrée reconnaît un état Haut,
- si nous avons entre -0.3V et 0.8V, l'entrée reconnaît un état Bas.

Ainsi, notre signal d'entrée doit être généré dans cette gamme de tension. Pour cela, nous utilisons par exemple un bouton poussoir, qui commande un inverseur qui ferme le contact pour une mise à 3.3V, la tension d'alimentation. Voir figure 13 page 28. Le fonctionnement est le suivant :

- lorsque le poussoir BP n'est pas pressé, le contact est ouvert, et la tension  $u(t)$  est à 0V,
- lorsque le poussoir est pressé, le contact est fermé, et la tension  $u(t)$  passe de 0 à 3,3V.

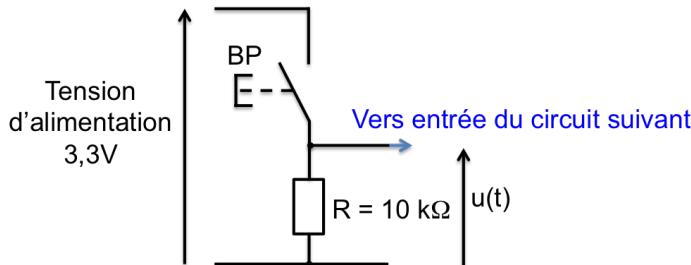


FIGURE 13 – Cablage du bouton poussoir, l'appui provoque le passage à 3,3V

Sur la platine de connexions, un bouton poussoir est connecté sur l'entrée 25 selon le schéma de la figure 13.

```
1 >>> bp = Pin(25, Pin.IN)
2 >>> bp.value()      # lit et retourne la valeur de l'objet bp
```

►Q.41: Vérifier que l'instruction en ligne 2 retourne 0 ou 1, en fonction du fait que vous appuyez ou pas sur le bouton poussoir lors de l'exécution. Pour cela, saisissez plusieurs fois l'instruction, ... ou lire la question suivante.

►Q.42: Afin d'éviter de saisir une instruction plusieurs fois, le raccourci clavier est, lorsque l'on est dans la console, les touches «flèche haut» et «flèche bas», qui permettent de remonter ou redescendre dans l'historique des instructions.

►Q.43: Connecter un bouton poussoir sur l'entrée 34. Pour cela, on utilise un connecteur «GROVE», voir figure 14 page 28, dont les fils sont repérés comme ceci :

- Noir : Masse.
- Rouge : 3,3V.
- Blanc : fil non connecté.
- Jaune : tension qui passe de 0 à 3,3V lors de l'appui.

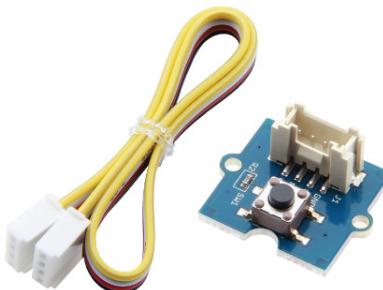


FIGURE 14 – Bouton poussoir Grove, avec son câble

```
1 >>> bpGrove = Pin(34, Pin.IN)
2 >>> bpGrove.value()      # lit et retourne la valeur de l'objet bp
```

►Q.44: Vérifier que l'instruction en ligne 2 retourne 0 ou 1, en fonction du fait que vous appuyez ou pas sur le bouton poussoir Grove.

## 5.2 Création d'un script

### 5.2.1 Faire clignoter une LED en continu - Boucle while

Nous allons faire clignoter la LED bleue connectée à la sortie 2.

10 – led.py - Fait clignoter une LED à 1 Hz ; solution 1

```
1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5
6 while True:
7     led.value(1)          # on allume la LED
8     time.sleep(0.5)       # délai de 0.5 s
9     led.value(0)          # on éteint la LED
10    time.sleep(0.5)      # délai de 0.5 s
11
12 #while True:
13 #    if led.value() == 1 :           # si la LED est allumée...
14 #        led.value(0)             # on l'éteint
15 #    else :                      # sinon (elle est éteinte)...
16 #        led.value(1)             # on l'allume
17 #    time.sleep(0.5)            # délai de 0.5 s entre 2 boucles (1 Hz)
18
19 #while True:
20 #    led.value(not led.value()) ; # inversion de l'état de la LED
21 #    time.sleep(0.5)            # délai de 0.5 s entre 2 boucles (1 Hz)
```

Les lignes 6 à 10 sont faciles à comprendre ; on allume ou on éteint la LED, et entre chaque changement on attend 0,5s, ce qui fait bien une période de 1s, soit une fréquence de 1 Hz.

**Variante 1 :** Les lignes 12 à 17 sont mises en commentaires. Elles réalisent à peu près la même chose que les lignes 6 à 10. En ligne 13, on teste si la LED est allumée à l'aide de l'instruction `led.value()` qui commande une lecture de l'état de la LED, ou plutôt de la pin qui pilote la LED. Si la pin est à l'état «haut», la lecture retourne la valeur «1». On teste si la lecture vaut 1, et si oui on éteint la LED avec l'instruction `led.value(0)`, et sinon on l'allume avec l'instruction `led.value(1)`. Et ensuite, on attend 0,5 s avant de recommencer le cycle en revenant à la ligne 12.

►Q.45: Tester ce code. Attention, pour ce test, le plus simple est de permuter les mises en commentaires des lignes 12-17 et 6-10.

►Q.46: Sortir du programme ; pour cela, nous n'avons pas d'autre possibilité que de l'interrompre en cliquant sur le bouton «Stop» de Thonny.

►Q.47: Vérifier que nous pouvons simplifier en ligne 13 en écrivant simplement : `if led.value()` :

**Variante 2 :** Les lignes 19 à 21 sont mises en commentaires. Elles réalisent la même fonction que les lignes 12 à 17. Cette écriture signifie que l'on écrit, en ligne 20, à chaque itération, la valeur actuelle lue, mais *inversée* ; nous changeons donc bien d'état à chaque itération.

►Q.48: Vérifier cela en permutant les mises en commentaires des lignes 19-21 et 12-17.

### 5.2.2 Faire clignoter une autre sortie que la sortie 2

►Q.49: Connecter la LED Grove sur la sortie 18, et modifier le code pour faire clignoter cette LED. Les fils sont repérés comme ceci :

### 5.2.3 Faire clignoter la LED un certain nombre de fois - L'instruction FOR

Décidons maintenant d'effectuer 6 changements d'état, soit 3 cycles complet allumage/extinction, chaque cycle durant une seconde (2 fois 0,5 s). Le programme dure 3 secondes. Le code est alors :

11 – led-3-fois.py - Fait clignoter 3 fois une LED, puis s'arrête

```
1 from machine import Pin
```

```

2 import time
3
4 led = Pin(2, Pin.OUT)
5
6 for a in range (6) :
7     led.value(not led.value())
8     time.sleep(0.5)

```

La variable `a` prendra successivement toutes les valeurs entières définies par `range(6)`, soit les 6 valeurs de 0 à 5. Donc nous avons bien réalisé 3 cycles complets ON-OFF.

►Q.50: Vérifier le bon fonctionnement.

#### 5.2.4 Mesure de temps

Nous allons, par exemple, mesurer le temps dont a besoin MicroPython pour allumer et éteindre la LED 10 000 fois, le plus rapidement possible, donc sans délai entre chaque état. Nous afficherons le temps total pour les 10 000 cycles, et le temps pour un seul cycle (en  $\mu\text{s}$ ). Nous ferons ainsi :

**12 – led-temps.py** - Fait clignoter 10 000 fois une LED, mesure le temps

```

1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5
6 start = time.ticks_ms()           # memorise la valeur du ticks au lancement
7
8 for iteration in range (10000) :
9     led.value(1) ; led.value(0)
10
11 delta = time.ticks_ms() - start # calcule l'écart : après - avant
12
13 print ("Temps total pour 10 000 cycles :" , delta, "ms")
14 print ("Temps pour 1 cycle ON/OFF      :" , delta/10, "microsecondes")

```

►Q.51: Vérifier que le temps est assez court, en tout cas le clignotement est assez rapide pour ne pas être détectable à l'oeil. Nous voyons juste la LED s'allumer environ à la moitié de sa luminosité maxi, pendant moins d'une seconde, environ 200 ms.

En réalité, la LED est allumée durant la moitié de sa période, puis elle est éteinte durant l'autre moitié, et cela 10 000 fois durant ces 200 ms environ.

Nous remarquons en ligne 9 que plusieurs instructions peuvent se trouver sur la même ligne ; il faut juste les séparer par le signe «`;`». Cela peut être utile parfois, même si la lisibilité en est réduite. Cela est à éviter en général dans les programmes. Pourtant, dans cet ouvrage, j'utiliserais parfois cette manière de procéder, pour écouterer la longueur du code, lorsque cela n'est pas gênant pour la lisibilité.

#### 5.2.5 Considérations/rappels sur les bibliothèques - et sur le module time

Par défaut, avant l'importation, Python n'a pas accès à la fonction `sleep` car elle est incluse dans le module `time`. Cette fonction `sleep` permet de temporiser, d'attendre, un temps exprimé en secondes. Nous avons importé tout ce module `time`, donc il est nécessaire de définir quelle fonction nous appellerons précisément, et cela se fait par `time.sleep()`.

En réalité, quand nous avons tapé `import time`, cela a créé un espace de noms dénommé `time`, contenant les variables et fonctions du module `time`, et ainsi cela rend accessible la fonction `sleep`.

**En résumé sur les modules :** Pour importer un module, même s'il fait partie de la distribution de MicroPython, il est nécessaire de le signifier avec l'instruction `import` de manière à le rendre accessible. Pour importer un module qui est déjà chargée dans la mémoire de l'ESP32, on opère de la même manière. Nous verrons cela plus loin lors de l'importation d'un module que nous créerons, et que nous chargerons en mémoire.

Nous pouvons aussi importer d'un seul coup toutes les fonctions du module `time`, et il aurait alors fallu écrire pour cela :

```
from time import *
...
sleep(0.5)
```

Cela nous permet d'éviter d'écrire `time.sleep`, mais directement `sleep`.

►Q.52: Tester cela.

Mais nous pouvons aussi, de manière plus explicite, importer seulement la fonction `sleep`. Voilà donc finalement la manière la plus élégante de faire, car nous précisons explicitement quelle fonction nous voulons utiliser, au lieu de toutes les importer :

```
from time import sleep
...
sleep(0.5)
```

►Q.53: Tester cela.

### 5.2.6 Revenons sur la lecture d'une entrée numérique

Nous n'avons utilisé jusqu'à maintenant que des sorties numériques. Nous avons ainsi pu allumer une LED ou commander n'importe quel système qui est capable de reconnaître (ou recevoir) un état logique et agir en conséquence...

Mais nous pouvons aussi lire l'état d'une entrée, ce qui permet de déterminer si un bouton a été pressé ou pas (le bouton pressé générera dans notre cas un niveau «Haut»). La lecture se fait, dans l'exemple qui suit, à la fréquence de 10 lectures par seconde, car nous attendons 0,1 s entre deux itérations (voir ligne 12).

L'exemple suivant allume une LED si le bouton est pressé. A chaque itération, nous lisons (ligne 7 à 11) l'état du bouton, et s'il est pressé, on allume, sinon on éteint.

Une autre manière de faire, sur les trois lignes commentées (lignes 14 à 16), est de lire l'état du bouton par `bouton.value()`, et d'affecter directement le résultat à la sortie 2 avec l'instruction `led.value(bouton.value())`, ce qui allume la LED s'il est «True».

►Q.54: Un bouton poussoir est déjà connecté sur l'entrée 25 . Vérifier le fonctionnement du programme, avec les deux manières de faire.

#### 13 – led-bouton.py - Allume une LED si un bouton est pressé

```
1 from machine import Pin
2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bouton = Pin(25, Pin.IN)
6
7 while True :
8     if bouton.value() :
9         led.value(1)
10    else :
11        led.value(0)
12    sleep(0.1)
13
14 #while True :
15 #    led.value(bouton.value())
16 #    sleep(0.1)
```

### 5.2.7 Arrêt du programme par un bouton poussoir

Normalement, un programme, quand il est lancé, ne s'arrête que lorsque l'utilisateur le lui demande, à l'aide d'un bouton «Quitter» ou «Stop» par exemple.

Mais, pour faciliter et simplifier les choses, nous allons faire en sorte que le programme s'exécute durant un temps juste suffisant pour constater son bon fonctionnement, ou prévoir une sortie du programme par un appui sur un poussoir physique de notre montage par exemple.

Dans le code qui suit, la condition d'arrêt sera l'appui sur le bouton poussoir connecté à la borne 25.

#### 14 – led-stop.py - Fait clignoter une LED à 1 Hz avec STOP par bouton

```
1 from machine import Pin
```

```

2 import time
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 while not bp.value() :           # continue tant que bp.value() == 0
8     led.value(not led.value())
9     time.sleep(0.5)

```

**Attention :** Dans ce code, la lecture du poussoir a lieu toutes les 0,5 seconde, il est donc possible que la lecture ne soit pas réalisée si l'appui ET le relâchement ont eu lieu tous les deux pendant l'attente entre deux lectures (ligne 9). Dans ce cas, appuyer un peu plus longtemps, pour être assuré que votre appui soit lu lors du passage par la ligne 7.

►Q.55: Tester en appuyant brièvement (fonctionnement aléatoire), puis longuement (fonctionnement correct) sur le poussoir.

### 5.3 Application : un temporisateur d'éclairage

Voyons maintenant un exemple qui résume tout ce que nous avons vu jusqu'à présent : un temporisateur d'éclairage. Ce temporisateur permet, lors de l'appui sur un bouton (pin 25), d'allumer pendant 2 secondes une lumière, simulée par la LED connectée sur la pin 2. A chaque appui sur le bouton, on affecte au compteur `compt` la valeur 20. A chaque itération, qui dure 0.1 s, le compteur est décrémenté. La lumière n'est allumée que si la valeur du compteur est positive, soit pendant  $20 * 0,1s = 2s$ .

#### 15 – tempo-eclairage.py - Temporisateur d'éclairage

```

1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 compt = 0
8 while True :
9     if bp.value() : compt = 20
10    if compt > 0 :
11        led.value(1)
12    else :
13        led.value(0)
14    time.sleep(0.1)
15    compt = compt - 1

```

►Q.56: Vérifier que l'appui sur le bouton, même si la lumière est déjà allumée, relance un cycle d'éclairage pour une durée de 2 s.

Pour comprendre en détail ce qu'il se passe, nous utilisons la fonction `print` pour suivre les valeurs du compteur, et l'état de la lumière. Nous constaterons que le compteur peut prendre des valeurs négatives, ce qui n'est pas gênant en première approche.

#### 16 – tempo-eclairage-print.py - Temporisateur d'éclairage avec affichage des valeurs

```

1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 compt = 0
8 while True :
9     if bp.value() : compt = 20 ; print ("Light ON")
10    print(compt)
11    if compt > 0 :
12        led.value(1)
13    else :

```

```

14     led.value(0)
15     time.sleep(0.1)
16     compt = compt - 1
17     if compt == 0 : print ("Light OFF")

```

►Q.57: Vérifier que «Light ON» s'affiche quand on appuie sur le poussoir, et s'affiche à nouveau toutes les 0,1 s tant que le poussoir est maintenu pressé.

Voici maintenant quelques simplifications d'écriture, et une réduction du code à son essentiel :

### 17 – tempo-eclairage-optimise.py - Temporisateur d'éclairage optimisé

```

1 from machine import Pin
2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 compt = 0
8 while True :
9     if bp.value() : compt = 20
10    led.value(compt > 0)      # si compt > 0, cela fait led.value(True)
11    sleep(0.1)
12    compt -= 1              # raccourci d'écriture pour compt = compt - 1

```

►Q.58: Arrêter le déroulement du programme par le bouton «Stop» de Thonny.

## 5.4 Flasher le code dans la carte

Notre temporisateur, pour le moment, doit être relié à l'ordinateur, ou une carte Raspberry qui le pilote, pour pouvoir fonctionner ; le code est seulement stocké dans l'IDE Thonny, et n'est jamais stocké sur la carte. La carte ESP32 reçoit seulement les instructions à réaliser, ou des morceaux de script, dans sa mémoire vive (RAM). L'ordinateur doit alors rester connecté à la carte, car il effectue le transfert des infos, et peut afficher, nous l'avons vu, les résultats avec la fonction `print`.

Cependant, nous souhaitons maintenant que le fonctionnement soit autonome, de manière à ce que l'ESP32 puisse être installé dans un endroit approprié pour sa fonction, c'est-à-dire dans une cage d'escalier par exemple, relié à un bouton du type interrupteur «Legrand», et qui commande un relais électromagnétique pour allumer ou éteindre un vraie ampoule à visser, du type 230V-10W à LED et culot E27, installé dans un luminaire fixé au plafond.

Nous allons donc envoyer, ou autrement dit «téléverser», le code dans la mémoire Flash (mémoire maintenue même hors tension) de la carte. Ce code sera téléversé comme étant le script principal, le `main.py`, c'est-à-dire comme celui qui doit être exécuté lorsque la carte est mise sous tension, et qu'elle n'est pas connectée à l'ordinateur et sous son contrôle.

**Procédure :** Sélectionner l'onglet «File», choisir «Save As», puis «MicroPython device», et enregistrer sous le nom `main.py`.

- Lorsque vous utilisez le nom `main.py`, alors à chaque démarrage de la carte, elle exécutera ce code là, en commençant s'il existe par le code contenu dans le `boot.py`, qui est utilisé pour initialiser le système.
- Si vous utilisez un autre nom, ce fichier sur la carte pourra seulement être utilisé par le fichier `main.py` en tant que ressource, pour y stocker des modules, des fonctions, des classes, des variables.

A partir de maintenant, le code est chargé en mémoire Flash, et il y restera même si la carte n'est plus sous tension. Il est alors possible de déconnecter l'ordinateur, et le code est exécuté si on alimente simplement la carte par une alimentation via l'USB ou une alimentation dédiée si la carte en est pourvue.

►Q.59: Connectez un chargeur de smartphone, pour constater que notre temporisateur fonctionne correctement.

►Q.60: Connecter le relais, et vérifier le bon fonctionnement.

## 6 Les sorties PWM

### 6.1 Retour sur le principe de la sortie numérique

Une sortie numérique génère au niveau Haut une tension d'environ 3,3V, et au niveau Bas une tension d'environ 0V. Il est possible d'obtenir d'autres niveaux de tension, c'est à dire toute valeur entre 0 et 3,3V, mais seulement avec des sorties analogiques, mais ce n'est pas l'objet de cette section. Voyons plutôt la sortie numérique PWM, qui permet de réaliser quelque chose de très similaire au comportement de la sortie analogique, et qui est plus souvent utilisée.

### 6.2 PWM pour utilisation avec une LED

Une sortie PWM est une sortie numérique, dont l'état de la sortie change à une haute fréquence (en général entre 100 et 40 000 Hz), de sorte que l'œil ne perçoivent qu'une valeur moyenne. Il est donc possible, avec seulement un état allumé et un état éteint, d'obtenir toutes les valeurs d'éclairement intermédiaire de 0 à 100%, pour des valeurs d'entrée de 0 à 1023, selon la figure suivante :

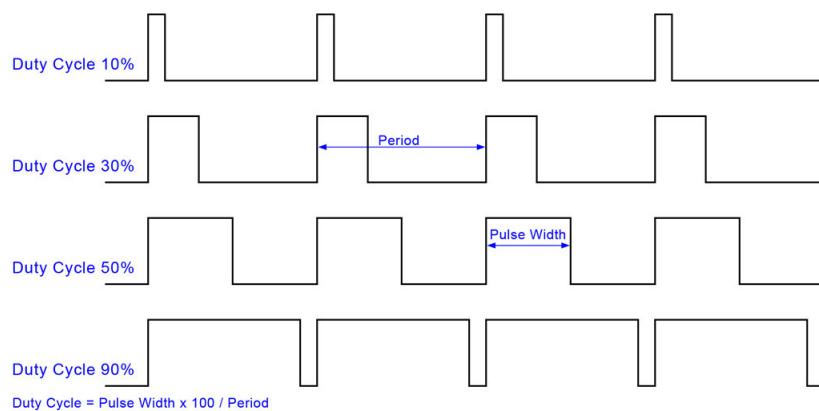


FIGURE 15 – Signal PWM généré par la fonction AnalogWrite, et rapport cyclique de 10% à 90%

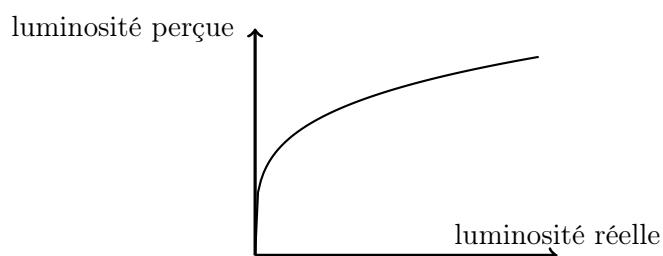
Le rapport cyclique de 100% est obtenu avec la valeur maximale 1023.

#### 6.2.1 Avec la console

Les instructions suivantes sont saisies directement dans la console.

```
>>> from machine import Pin, PWM  
>>> frequence = 500  
>>> led = PWM(Pin(2), frequence)    # initialise la fréquence du PWM  
>>> led.duty (1023)
```

- Q.61: Faire varier la valeur du rapport cyclique, par exemple mettre 1023, 100, 10, 1.
- Q.62: Vérifier que la luminosité diminue.
- Q.63: Vérifier aussi que l'impression de lumière n'est pas proportionnelle au rapport cyclique. En réalité, notre œil réagit à peu près en fonction du logarithme ou de la racine carrée de la luminosité, c'est à dire qu'une lumière 10 fois plus forte (rapport cyclique 10 fois plus grand) sera perçue comme seulement 2 à 3 fois plus forte. Revérifier si besoin avec les valeurs 1023 et 100.



## 6.2.2 Cr ation de trois cycles de mont e

Pour v rifier la non lin arit , nous faisons croitre le rapport cyclique de 0   1023, par pas de 10, toutes les 10 ms, donc en environ une seconde, et cela trois fois.

18 – pwm.py - PWM - 3 cycles de mont e

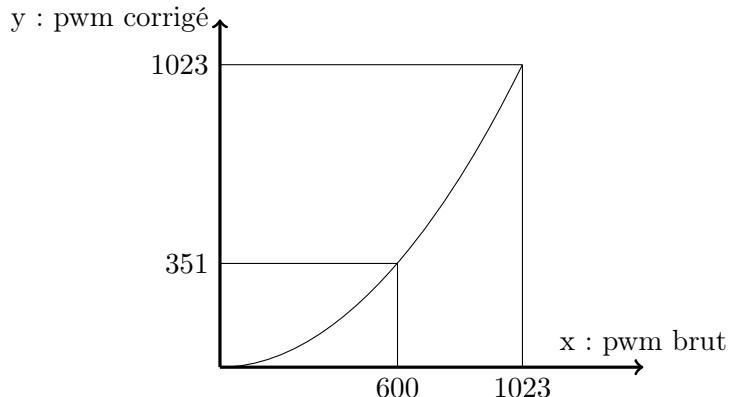
```
1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # Initialise la freq : 500 Hz
5
6 for boucle in range(3) :
7     for i in range(0, 1023, 10) :    # de 0   1023 par pas de 10
8         led.duty(i)
9         sleep_ms(10)
10    led.deinit()                  # lib ration des ressources du PWM
```

La mont e de 0   1023 est lin aire, mais   cause de la non-lin arit  de la perception de la luminosit , il nous semble que la mont e en luminosit  semble rapide ; en effet, la LED semble  tre rapidement allum e   la moiti  du temps, puis semble rester allum e   pleine puissance, alors qu'en r alit  le rapport cyclique continue d'augmenter.

►Q.64: V rifier cela.

## 6.2.3 Prise en compte du caract re non lin aire de l' eil

Nous allons corriger cela en partie, en cr ant une progression non lin aire de la luminosit , et par exemple, pour simplifier, une progression parabolique. Le rapport cyclique brut sera not   $x$ , et le rapport cyclique corrig  sera not   $y$ . Ainsi, nous aurons un rapport cyclique corrig   $y = 0$  pour un rapport cyclique brut  $x = 0$ ; de m me nous aurons  $y = 1023$  pour  $x = 1023$ , mais nous aurons une progression parabolique de la forme  $y = x^2$  pour toute valeur de  $x$  entre 0 et 1023.



►Q.65: Comprendre que la mani re de faire cela peut  tre la suivante :

- diviser  $x$  par 1023, pour avoir  $x$  variant lin airement de 0   1,
- prendre le carr  du r sultat pour avoir une parabole, variant aussi de 0   1,
- et multiplier par 1023 pour retrouver notre parabole variant de 0   1023.

Ainsi, nous aurons :  $y = (x/1023)^2 * 1023$

►Q.66: Calculer la valeur de  $y$  pour  $x = 600$ .

19 – calcul-correction.py - Calcul correction luminosit 

```
1 for valeur_brute in [0, 200, 400, 600, 800, 1023] :
2     valeur_corrigee = int((valeur_brute/1023)** 2 * 1023)
3     print('valeur brute : {0:4d} ; valeur corrig e : {1:4d} '.format
4           (valeur_brute , valeur_corrigee))
```

Dans ce code, nous prenons les 7 valeurs de la liste, de 0   1023, pour v rifier que nous obtenons apr s calcul une parabole passant par  $(0; 0)$  et  $(1023; 1023)$ , avec son sommet en 0. Nous obtenons le r sultat ci-dessous :

```

valeur brute :    0 ; valeur corrigée :    0
valeur brute :  200 ; valeur corrigée :   39
valeur brute :  400 ; valeur corrigée :  156
valeur brute :  600 ; valeur corrigée :  351
valeur brute :  800 ; valeur corrigée :  625
valeur brute : 1023 ; valeur corrigée : 1023

```

►Q.67: Comprendre aussi que nous avons arrondi les valeurs corrigées avec la fonction `int`, car la méthode `duty` attend des valeurs entières, sinon une erreur serait générée.

►Q.68: Remarquer que nous avons utilisé une nouvelle syntaxe pour l'affichage; celle-ci nous permet d'aligner les chiffres à droite en indiquant 4 décimales. Le `{0:4d}` et le `{1:4d}` sont donc le premier (0) et le deuxième élément (1) de la liste de valeurs à afficher, avec 4 chiffres affichés, et indiqués à l'endroit où ils vont apparaître dans la chaîne de caractère. Les valeurs seront données par la «méthode» `format`.

Maintenant, nous allons utiliser cette correction pour le PWM.

## 20 – pwm-correction.py - PWM avec correction luminosité

```

1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # Initialise la freq a 500 Hz
5
6 for boucle in range(3) :
7     for i in range(0, 1023, 10) :
8         led.duty(int((i/ 1023)** 2 * 1023))
9         sleep_ms(10)
10 led.deinit()

```

►Q.69: Vérifier que la montée en luminosité se fait de manière beaucoup plus linéaire.

Le coefficient 2 est donc en première approximation une valeur correcte. Mais peut être que nous pourrions l'ajuster. En effet, est-ce que le coefficient 3 permettrait une progression «ressentie» plus linéaire? Une solution serait de tester le programme avec cette nouvelle valeur. Mais nous allons plutôt demander, au cours de l'exécution du code, à l'utilisateur quelle valeur de coefficient il souhaite tester :

## 21 – pwm-correction-input.py - PWM avec correction luminosité choisie

```

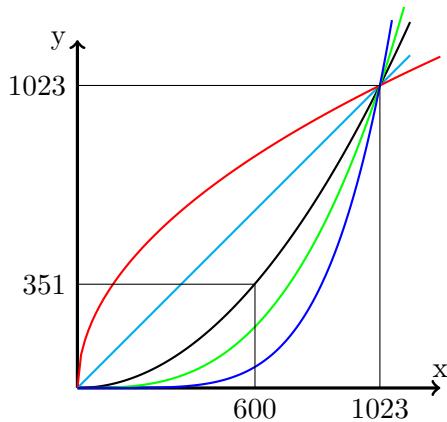
1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # Initialise la freq a 500 Hz
5
6 coef = float(input('Coefficient : '))
7
8 for boucle in range(3) :
9     for i in range(0, 1023, 10) :
10        led.duty(int((i/1023)** coef * 1023))
11        sleep_ms(10)
12 led.deinit()

```

La fonction `input()` demande à l'utilisateur, par l'intermédiaire du Shell, quelle valeur il choisit pour son coefficient de correction. Cette valeur renournée est une chaîne de caractère, que l'on transforme en nombre flottant (à virgule) par la fonction `float`.

►Q.70: Vérifier les valeurs 2, 3, 10, puis 0.5, et toute valeur réelle que vous souhaitez, entre 0.1 et 10.

Sur le graphe ci dessous sont tracées les courbes pour un coefficient de 2 (en noir), 3 (en vert), 5 (en bleu), et aussi 1 (en cyan), et 0.5 (en rouge, on aura reconnu une fonction racine carrée).



Choisir la valeur de coefficient que vous préférez. Nous voyons en ligne 11 du code suivant que j'ai choisi le coefficient 3.5.

Maintenant, nous allons créer la fonction `correction()` pour simplifier l'écriture, et permettre une réutilisation aisée dans un autre code.

## 22 – pwm-fonction-correction.py - PWM avec correction luminosité par fonction

```

1  from time import sleep_ms
2  from machine import Pin, PWM
3
4  led = PWM(Pin(2, Pin.OUT), 500)      # initialise la freqence a 500 Hz
5
6  def correction(i, coef) :
7      return int ((i / 1023) ** coef * 1023)
8
9  for boucle in range(3) :
10     for i in range(0, 1023, 10) :
11         led.duty(correction(i, 3.5)) # utilisation fonction correction
12         sleep_ms(10)
13 led.deinit()

```

## 7 Utilisation du ruban de LED NeoPixel - Première approche

### 7.1 Mélange de couleurs

Incorporer de nombreuses LEDs dans un projet électronique peut compliquer sa réalisation, avec un code difficile à maintenir. Mais l'arrivée de LEDs disposant d'une puce pilote intégrée change radicalement la donne en allégeant le travail du microcontrôleur et le câblage. Elle permet aussi de se concentrer sur le code. Le WS2812, avec sa source de lumière intégrée, ou plus communément nommé «NeoPixel» par la société «Adafruit», est la dernière avancée dans la quête pour obtenir des LEDs pleine de couleurs, simple à mettre en oeuvre, évolutive et bon marché.

Les LEDs rouge, vert et bleue sont intégrées côté à côté sur un petit support CMS (composants montés en surface), lui-même intégré sur la puce du pilote (le contrôleur), le tout contrôlé par un simple fil. Elles peuvent être utilisées individuellement, arrangées pour former une longue chaîne, ou assemblées pour réaliser des formes intéressantes.

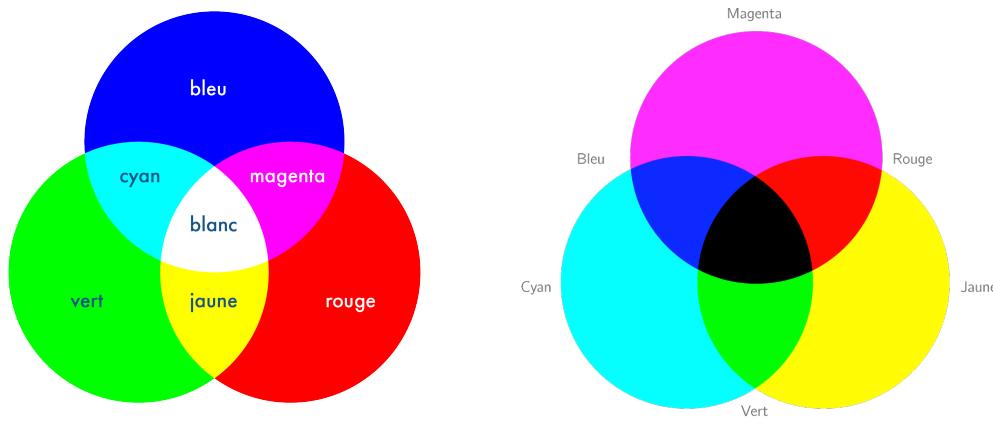
La couleur finale obtenue est l'addition des trois couleurs primaires, c'est-à-dire le rouge, le vert, et le bleu. Voir figure 16 page 38.

### 7.2 Premier essai

Le réglage de la luminosité (sur le principe du PWM vu précédemment) se fait ici de 0% à 100%, pour un réglage de 0 à 255 (et non plus 1023 comme précédemment). Ainsi, pour une valeur de 30, on aura 30/255 de la pleine luminosité (hors prise en compte de la non linéarité de l'œil ...)

►Q.71: Tester le programme suivant pour comprendre :

## 23 – neopixel.py - Test du ruban NeoPixel



(a) En synthèse additive : rouge, vert, bleu

(b) En synthèse soustractive : magenta, cyan, jaune

FIGURE 16 – Les couleurs primaires

```

1 from machine import Pin, ADC
2 from neopixel import NeoPixel # import de la classe NeoPixel du module
3
4 n = 8                         # nombre de pixels
5 p = 26                         # pin de commande du neopixel
6 np = NeoPixel(Pin(p), n)        # creation de l'instance np
7
8 np[0] = (30, 0, 0)             # rouge, 30/255 brightness, soit 12%
9 np[1] = (255, 0, 0)            # rouge, 100% brightness
10 np[2] = (0, 255, 0)           # vert, 100% brightness
11 np[3] = (0, 0, 255)           # bleu, 100% brightness
12 np[4] = (255, 255, 0)          # rouge + vert = jaune
13 np[5] = (0, 255, 255)          # vert + bleu = cyan
14 np[6] = (255, 0, 255)          # rouge + bleu = magenta (violet)
15 np[7] = (255, 255, 255)          # rouge + vert + bleu = blanc
16 np.write()                    # Ecriture des valeurs sur le ruban

```

Pour bien comprendre que chaque élément du NéoPixel est en fait composée de 3 LEDs, voici un code avec des luminosités plus faibles qui permettent de bien voir les différentes composantes.

### 7.3 Deuxième essai pour voir les 3 composantes

#### 24 – neopixel-test.py - Test du ruban NeoPixel - Deuxième essai

```

1 from machine import Pin, ADC
2 from neopixel import NeoPixel # import de la classe NeoPixel du module
3
4 n = 8                         # nombre de pixels
5 p = 26                         # pin de commande du neopixel
6 np = NeoPixel(Pin(p), n)        # creation de l'instance np
7
8 np[0] = (10, 0, 0)              # rouge, 10/255 brightness, soit 4%
9 np[1] = (0, 10, 0)              # vert, 10/255 brightness, soit 4%
10 np[2] = (0, 0, 10)             # bleu, 10/255 brightness, soit 4%
11 np[3] = (10, 10, 10)            # blanc = rouge + vert + bleu
12 np[4] = (50, 0, 0)              # rouge, 50/255 brightness, soit 20%
13 np[5] = (0, 50, 0)              # vert, 50/255 brightness, soit 20%
14 np[6] = (0, 0, 50)              # bleu, 50/255 brightness, soit 20%
15 np[7] = (50, 50, 50)            # blanc = rouge + vert + bleu
16 np.write()                    # Ecriture des valeurs sur le ruban

```

Maintenant, nous allumons toutes les LED, puis nous les éteignons, dans le même ordre.

#### 25 – neopixel-defilement.py - Défilement des LED

```

1 from machine import Pin, ADC
2 from time import sleep
3 from neopixel import NeoPixel # import de la classe NeoPixel du module
4
5 n = 8 # nombre de pixels
6 p = 26 # pin de commande du neopixel
7 delai = .125
8 np = NeoPixel(Pin(p), n) # creation de l'instance np
9
10 for x in range(0, n):
11     np[x] = (0, 0, 255) # bleu, 100% brightness
12     np.write()
13     sleep(delai)
14
15 for x in range(0, n):
16     np[x] = (0, 0, 0) # eteint
17     np.write()
18     sleep(delai)

```

Maintenant, nous allons allumer  $x$  pixels,  $x$  étant choisi par l'utilisateur. On allumera depuis la LED 0 jusqu'à la LED  $x - 1$ . Pour comprendre ce code, prenons par exemple la valeur de  $x = 2$ . Dans la boucle **for** la variable **led** prend les valeurs de 0 à 7 :

- lorsque led = 0, on a  $0 < 2$ , donc la led 0 est allumée,
- lorsque led = 1, on a  $1 < 2$ , donc la led 1 est allumée,
- lorsque led = 2, on n'a pas  $2 < 2$ , donc la led 1 n'est pas allumée,
- lorsque led > 2, les led sont éteintes.

Donc nous avons bien 2 LEDs allumées.

## 7.4 L'utilisateur choisit le nombre de LED allumées

**26 – neopixel-x-allumes.py** - L'utilisateur choisit d'allumer x LED

```

1 from machine import Pin, ADC
2 from time import sleep
3 from neopixel import NeoPixel
4
5 n = 8 # nombre de pixels
6 p = 26 # pin de commande du neopixel
7 np = NeoPixel(Pin(p), n) # creation de l'instance np
8
9 x = int(input("Combien voulez vous de LED allumees ? : "))
10
11 for led in range(0, n):
12     np[led] = (0, 0, 20*(led<x)) # = 255 si (led < x) ; = 0 sinon
13 np.write()

```

►Q.72: Modifier le code pour allumer en vert les autres LED au lieu de les laisser éteintes.

## 7.5 Tirage aléatoire

Maintenant, nous allons effectuer un tirage aléatoire d'un nombre compris entre 0 et 8, et afficher le résultat sur le NeoPixel.

►Q.73: Vérifier le bon fonctionnement.

**27 – neopixel-random.py** - On allume x LED de manière aléatoire

```

1 from machine import Pin, ADC
2 from time import sleep
3 from neopixel import NeoPixel
4 from random import randint
5
6 n = 8 # nombre de pixels
7 p = 26 # pin de commande du neopixel
8 np = NeoPixel(Pin(p), n) # creation de l'instance np

```

```

9      x = randint(0,8)                      # tirage aleatoire entre 0 et 8
10     print(x)
11
12
13     for led in range(0, n):
14         np[led] = (0, 0, 50*(led<x))    # = 50 si (led < x) ; = 0 sinon
15     np.write()

```

Pour vérifier le bon fonctionnement de la fonction `randint`, nous allons effectuer un tirage de 20 valeurs, puis les classer, pour observer la répartition.

### 28 – tirage-aleatoire.py - Tirage de 20 valeurs entières entre 0 et 8

```

1 from random import randint
2
3 liste=[]
4
5 for i in range(20):          # pour 20 nombres...
6     tirage = randint(0,8)    # tirage entre 0 et 8
7     liste.append(tirage)    # ajout a la liste des tirages
8
9 print(liste)
10 liste.sort()                # classement de la liste dans l'ordre croissant
11 print(liste)

```

Et l'on obtient par exemple ceci :

```
[2, 0, 6, 6, 4, 4, 8, 8, 5, 2, 0, 5, 4, 7, 1, 1, 4, 3, 1, 7]
[0, 0, 1, 1, 1, 2, 2, 3, 4, 4, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8]
```

►Q.74: Vérifier le bon fonctionnement.

## 7.6 Arc-en-ciel

Et voici, pour finir en beauté ce TP!!

### 29 – neopixel-rainbow.py - Arc-en-ciel sur NéoPixel

```

1 from machine import Pin, ADC
2 from time import sleep_ms
3 from neopixel import NeoPixel
4
5 n = 8                      # nombre de pixels
6 p = 26                      # pin de commande du neopixel
7 np = NeoPixel(Pin(p), n)      # creation de l'instance np
8
9 def wheel(pos):
10     if pos < 0 or pos > 255:
11         return (0, 0, 0)
12     if pos < 85:
13         return (255 - pos * 3, pos * 3, 0)
14     if pos < 170:
15         pos -= 85
16         return (0, 255 - pos * 3, pos * 3)
17     pos -= 170
18     return (pos * 3, 0, 255 - pos * 3)
19
20 def rainbow_cycle(wait):
21     for j in range(255):
22         for i in range(n):
23             rc_index = (i * 256 // n) + j
24             np[i] = wheel(rc_index & 255)
25         np.write()
26         sleep_ms(wait)
27
28 while True :
29     rainbow_cycle(1)

```