

Table des matières

BUREAUTIQUE

PowerPoint

PROGRAMMATION EN PYTHON

Python

Python: Les outils pour développer

Thonny Python

Thonny Python - Installer des modules

Les variables en Python

Afficher ses variables avec print

Manipuler les variables en Python

Manipuler valeurs et variables: exemples guidés 🚀

Valeurs et variables: à toi de jouer 🧡

Valeurs et variables: à toi de jouer (2) 🧡

Bien nommer ses variables en Python

Python - Opérations mathématiques de base

Python: Les conditions

Python: Les conditions simples

Python: Opérations de comparaison

Python: Conditions / Exercices de base

Python: Les conditions (combinaisons)

Python - Les opérateurs logiques

Python: Conditions / Combinaisons / Exercices

Python - Les opérateurs logiques - Évaluation paresseuse

Python: Les listes

Python: Parcourir une Liste en Python

Python: Exercices sur les listes

Python: Les boucles

Python: Les boucles for

Python: Les boucles for (exercices 🧡)

Python: Compter et cumuler des valeurs dans une boucle

Python: Les boucles for sur des listes

Python: Afficher un menu de sélection

Python: Les fonctions

Python: Les fonctions (pratique autonome)

Python: Les fonctions (pratique guidée)

Python: Les modules

Introduction à Python

ELECTRONIQUE

Les actionneurs

Feux de circulation

Introduction aux buzzers

Utiliser un buzzer actif (KY-012)

Utiliser un buzzer passif (KY-006)

Les capteurs

BME280: pression atmosphérique et altitude

DHT11: Simulation du Capteur

DHT11: Capteur de température et d'humidité

HC-SR04: Capteur de distance à ultra-sons

KY-004: Bouton

Les micro-contrôleurs

Le Raspberry Pi Pico

Les Broches du Raspberry Pi Pico en MicroPython

MicroPython sur RPi Pico avec Thonny Python

Raspberry Pi Pico et les Ports GPIO

Utilisation du Raspberry Pi Pico avec MicroPython

Les différents types de signaux

CANSAT

Liens utiles

Méthodes agiles

La méthode Kanban



BUREAUTIQUE

PowerPoint

Le fichier se trouve en pièce jointe.



PROGRAMMATION EN PYTHON

Python

Python est un langage de programmation interprété, polyvalent et convivial qui a gagné en popularité au fil des ans. Créé par Guido van Rossum et publié pour la première fois en 1991, Python est devenu l'un des langages les plus utilisés dans le monde de la programmation en raison de sa simplicité, de sa lisibilité et de sa grande communauté de développeurs.

Les avantages de Python

► Simplicité et lisibilité

L'une des principales caractéristiques de Python est sa syntaxe simple et concise, qui le rend facile à apprendre et à utiliser même pour les débutants. La lisibilité du code Python est également un avantage majeur, permettant aux développeurs de comprendre rapidement le fonctionnement d'un programme.

► Polyvalence

Python est un langage polyvalent qui peut être utilisé dans une grande variété de domaines, notamment le développement web, l'analyse de données, l'intelligence artificielle, l'automatisation, la robotique et bien plus encore. Sa polyvalence en fait un outil indispensable pour de nombreux développeurs et entreprises.

► Grande bibliothèque standard

Python dispose d'une vaste bibliothèque standard qui offre des fonctionnalités prêtes à l'emploi pour diverses tâches de programmation. Cela permet aux développeurs d'économiser du temps en utilisant des modules et des packages déjà développés plutôt que de réinventer la roue.

► Communauté active

La communauté Python est l'une des plus grandes et des plus actives dans le monde de la programmation. Cette communauté dynamique contribue au développement continu de Python en créant de nouveaux packages, en fournissant des ressources d'apprentissage et en soutenant les nouveaux arrivants.

Qui utilise Python et pourquoi ?

► Développeurs web

Python est largement utilisé dans le développement web pour la création de sites web, d'applications web et de services web. Des frameworks populaires comme Django et Flask permettent de développer rapidement des applications web robustes et évolutives.

► Scientifiques des données

Python est devenu le langage de choix pour de nombreux scientifiques des données en raison de ses puissantes bibliothèques d'analyse de données telles que NumPy, Pandas et Matplotlib. Ces outils facilitent l'exploration, la manipulation et la visualisation des données.

► Ingénieurs en intelligence artificielle et en machine learning

Python est largement utilisé dans le domaine de l'intelligence artificielle et de l'apprentissage automatique en raison de sa facilité d'utilisation et de ses bibliothèques spécialisées telles que TensorFlow, Keras et scikit-learn. Ces bibliothèques permettent de développer et de déployer des modèles d'apprentissage automatique de manière efficace.

► Développeurs de logiciels et d'applications

De nombreuses entreprises utilisent Python pour le développement de logiciels et d'applications en raison de sa polyvalence, de sa productivité et de sa capacité à s'intégrer facilement avec d'autres langages et technologies.

La popularité croissante de Python

Python connaît une popularité croissante grâce à ses nombreux avantages et à sa polyvalence. Selon divers indices de popularité, tels que l'indice TIOBE et l'enquête Stack Overflow Developer Survey, Python figure parmi les langages de programmation les plus populaires et les plus demandés dans l'industrie.

Sa popularité continue de croître en raison de son adoption par de grandes entreprises comme Google, Facebook, Amazon et Netflix, qui utilisent Python pour diverses applications et services.

En conclusion, Python est bien plus qu'un simple langage de programmation. C'est un outil puissant et polyvalent qui continue de gagner en popularité grâce à sa simplicité, sa lisibilité, sa grande communauté et sa polyvalence. Que vous soyez un débutant en programmation ou un développeur expérimenté, Python offre une multitude d'opportunités pour créer des applications innovantes et résoudre des problèmes complexes dans divers domaines.

Notes de cours

- [Télécharger les notes de cours "officielles"](#)

EXERCICES DU JOUR

- [Exercices sur les variables](#)
- [Pour les courageux](#)
- [Exercices sur les conditions](#)
- [Exercices sur les boucles](#)

Vidéos "Python au lycée"

- Premiers pas: console, calculs, fichiers .py et print (6')
- Premiers pas: variables, commentaires (8'30)
- Les 4 types de variables de base
- Si...alors...(sinon) (partie 1) structure et condition (8'50)
- Entrée au clavier: input(), int() et float() (9'44)
- Si...alors...(sinon) (partie 2) (5'20)
- Booléens, comparaisons, random() (8')
- Nombres au hasard ("aléatoires") (5'40)
- Les fonctions (Partie 1) (5'38)
- Les fonctions (Partie 2) (5'38)
- Les boucles **for** (5'38)
- Les boucles **for... in range()** (5'38)

Python: Les outils pour développer

Lorsque vous vous lancez dans le développement en Python, choisir les bons outils peut faire toute la différence dans votre productivité et votre expérience de développement. Dans cet article, nous allons passer en revue certains des outils les plus populaires utilisés par les développeurs Python et discuter de leurs principales caractéristiques.

1. Visual Studio Code (VS Code)



Visual Studio Code est un éditeur de code léger et puissant développé par Microsoft. Il offre une multitude de fonctionnalités pour les développeurs Python, notamment :

- **Prise en charge de Python intégrée** : VS Code propose une prise en charge intégrée de Python avec des fonctionnalités telles que la coloration syntaxique, la saisie semi-automatique, le débogage et la gestion des environnements virtuels.
- **Extensions Python** : Vous pouvez étendre les fonctionnalités de VS Code en installant des extensions Python, telles que Python, Pylance, ou encore Anaconda Extension Pack, qui ajoutent des fonctionnalités avancées.
- **Personnalisable** : VS Code est hautement personnalisable avec des thèmes, des raccourcis clavier et des extensions pour répondre à vos besoins spécifiques de développement Python.

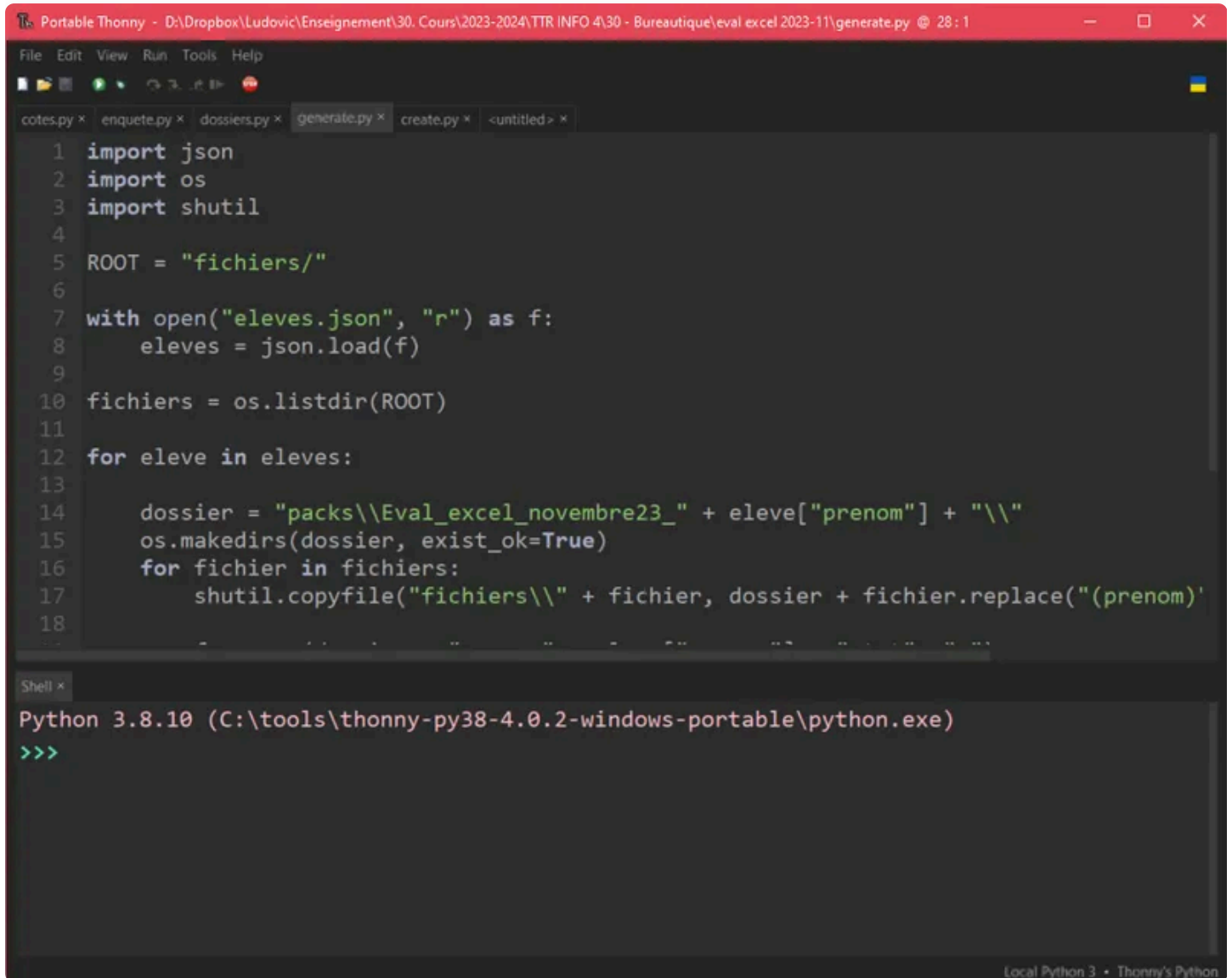
► Configuration de Visual Studio Code pour Python

Pour configurer VS Code pour le développement Python, suivez ces étapes simples :

1. Installez l'**extension Python** depuis le marché des extensions de VS Code.
2. Configurez votre environnement virtuel Python en sélectionnant l'interpréteur Python souhaité dans les paramètres de l'extension.
3. Utilisez les fonctionnalités intégrées de débogage et de gestion des packages pour développer efficacement vos projets Python.

Avant de pouvoir utiliser VS Code pour développer en Python, **vous devez d'abord avoir installé Python sur votre ordinateur.**

2. Thonny Python



```
Portable Thonny - D:\Dropbox\Ludovic\Enseignement\30. Cours\2023-2024\TTR INFO 4\30 - Bureautique\eval excel 2023-11\generate.py @ 28 : 1
File Edit View Run Tools Help
cotes.py x enquete.py x dossiers.py x generate.py x create.py x <untitled> x
1 import json
2 import os
3 import shutil
4
5 ROOT = "fichiers/"
6
7 with open("eleves.json", "r") as f:
8     eleves = json.load(f)
9
10 fichiers = os.listdir(ROOT)
11
12 for eleve in eleves:
13
14     dossier = "packs\\Eval_excel_novembre23_" + eleve["prenom"] + "\\\"
15     os.makedirs(dossier, exist_ok=True)
16     for fichier in fichiers:
17         shutil.copyfile("fichiers\\" + fichier, dossier + fichier.replace("(prenom)"
18
Shell x
Python 3.8.10 (C:\tools\thonny-py38-4.0.2-windows-portable\python.exe)
>>>
```

Thonny est un environnement de développement intégré (IDE) spécialement conçu pour les débutants en programmation Python. Ses caractéristiques principales incluent :

- **Interface utilisateur conviviale** : Thonny offre une interface utilisateur simple et intuitive qui facilite l'apprentissage de la programmation Python pour les débutants.
- **Débogage pas à pas** : Thonny propose une fonctionnalité de débogage pas à pas qui permet aux débutants de comprendre le comportement de leur code en le suivant ligne par ligne.
- **Explorateur de variables** : Thonny dispose d'un explorateur de variables qui affiche les valeurs des variables en temps réel pendant l'exécution du code.
- **Installation facile** : Thonny est facile à installer et est disponible pour Windows, macOS et Linux.

Un des avantages principaux de **Thonny**, c'est qu'il **est livré avec sa propre version de Python**. Il n'est donc pas nécessaire d'installer Python manuellement!

🔔 Pour tout savoir sur Thonny IDE et le télécharger, rendez-vous sur [la page Thonny Python](#)

3. www.online-python.com

[Online Python](https://www.online-python.com) est une plateforme en ligne qui offre un environnement de développement Python entièrement fonctionnel dans votre navigateur. Ses fonctionnalités comprennent :

- **Accès instantané** : Pas besoin d'installer quoi que ce soit. Il vous suffit d'ouvrir votre navigateur et de vous rendre sur le site pour commencer à coder en Python.
- **Prise en charge de la saisie semi-automatique** : Online Python propose une saisie semi-automatique qui facilite l'écriture de code Python.
- **Exécution de code en temps réel** : Vous pouvez exécuter votre code Python en temps réel et voir les résultats instantanément dans la sortie.
- **Partage de code** : Online Python vous permet de partager votre code avec d'autres personnes en générant simplement un lien.

Autres outils populaires pour le développement Python

En plus de Visual Studio Code, Thonny Python et www.online-python.com, il existe d'autres outils populaires utilisés par les développeurs Python pour améliorer leur flux de travail de développement. Parmi ces outils, on trouve :

► PyCharm

PyCharm est un environnement de développement intégré (IDE) développé par JetBrains, offrant une gamme complète de fonctionnalités pour le développement Python, y compris un débogueur intégré, la gestion des packages, la refonte de code et bien plus encore.

Il existe 2 versions de PyCharm: la version Pro (payante) et la version Community (gratuite), donc **fais attention à quelle version tu installes**.

► Jupyter Notebook

Jupyter Notebook est une application Web open source qui vous permet de créer et de partager des documents interactifs contenant du code Python, des visualisations et des explications textuelles. Il est largement utilisé pour l'analyse de données, l'apprentissage machine et la recherche scientifique.

► Comment installer Python sur son ordinateur

Installer Python sur votre ordinateur est un processus simple et direct. Voici les étapes générales pour installer Python :

1. Rendez-vous sur le site officiel de Python à l'adresse <https://www.python.org/downloads/> et téléchargez la dernière version de Python pour votre système d'exploitation (Windows, macOS, ou Linux).
2. Lancez le programme d'installation téléchargé et suivez les instructions à l'écran pour installer Python sur votre ordinateur. Assurez-vous de cocher l'option "Ajouter Python à PATH" lors de l'installation pour pouvoir exécuter Python depuis n'importe quel répertoire de votre système.
3. Une fois l'installation terminée, vous pouvez vérifier si Python a été correctement installé en ouvrant une fenêtre de terminal (ou d'invite de commande sur Windows) et en tapant la commande suivante :

```
python --version
```

Cette commande affichera la version de Python installée sur votre système. Si vous voyez le numéro de version de Python, cela signifie que l'installation s'est déroulée avec succès.

En suivant ces étapes simples, vous pouvez installer Python sur votre ordinateur et commencer à développer des applications en Python en un rien de temps.

Conclusion

Que vous soyez débutant ou développeur expérimenté, avoir les bons outils est essentiel pour développer en Python. Visual Studio Code, Thonny Python et www.online-python.com sont quelques-uns des outils les plus populaires et les plus utilisés dans la communauté Python. En choisissant l'outil qui convient le mieux à vos besoins et à votre style de développement, vous pouvez améliorer votre productivité et votre expérience de développement Python.

Thonny Python

Lorsqu'il s'agit d'apprendre à programmer en Python, choisir le bon environnement de développement peut faire toute la différence. Thonny IDE se présente comme un outil puissant et convivial, spécialement conçu pour les débutants en Python. Dans cet article, nous explorerons les avantages de Thonny, son interface utilisateur intuitive, et comment l'utiliser efficacement pour démarrer votre voyage dans le monde de la programmation Python.

Les Avantages de Thonny IDE

► Convivialité

Thonny se distingue par sa **simplicité d'utilisation**. Contrairement à certains IDE plus complexes, Thonny est conçu pour être **intuitif**, ce qui le rend **idéal pour les débutants** en programmation.

► Interface Unifiée

L'interface utilisateur de Thonny est soigneusement organisée, ce qui facilite la navigation et l'accès aux différentes fonctionnalités de l'IDE. Les outils essentiels tels que l'éditeur de code, la console interactive et le débogueur sont tous intégrés de manière transparente dans **une seule fenêtre**.

► Débogage Pas à Pas

Thonny offre un **débogueur intégré** qui permet aux utilisateurs d'exécuter leur code pas à pas, d'observer les valeurs des variables en cours d'exécution et de détecter les erreurs plus facilement.

► Gestionnaire de Packages Intégré

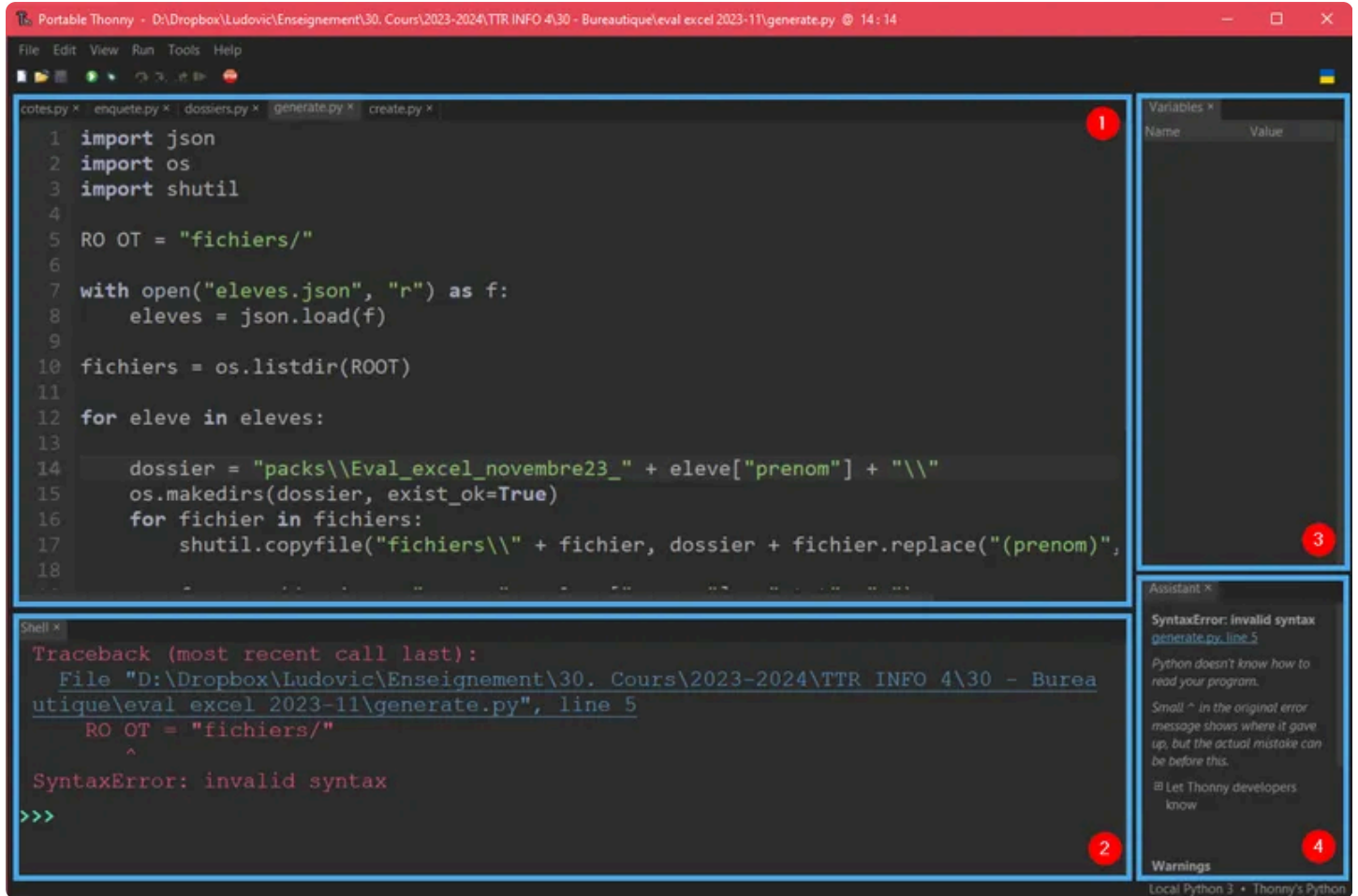
Thonny est livré avec un gestionnaire de packages/modules intégré qui facilite l'installation de modules Python supplémentaires directement depuis l'IDE, ce qui vous permet d'étendre les fonctionnalités de Python selon vos besoins.

► Documentation Intégrée

Une documentation complète est disponible directement dans Thonny, ce qui vous permet d'accéder rapidement aux informations sur les fonctions et les modules Python sans avoir à quitter l'IDE.

Interface de Thonny

L'interface utilisateur de Thonny est conçue pour être simple et intuitive, ce qui permet aux utilisateurs de se concentrer sur l'essentiel : écrire du code. Voici un aperçu des principaux composants de l'interface de Thonny :



► 1. Éditeur de Code

C'est là où vous écrivez et modifiez votre code Python. L'éditeur de code de Thonny offre des fonctionnalités telles que la **coloration syntaxique**, l'**indentation automatique** et la suggestion de code pour améliorer votre flux de travail de programmation.

💡 La **coloration syntaxique** est une fonctionnalité qui consiste à **attribuer des couleurs différentes** aux **différents éléments d'un langage de programmation**. Cette technique vise à rendre le code source **plus lisible et compréhensible** en mettant en évidence les différents éléments syntaxiques tels que les **mots-clés**, les **variables**, les chaînes de caractères, les commentaires, etc. La coloration syntaxique **aide les programmeurs à identifier rapidement la structure et la logique du code**, ce qui facilite la détection des erreurs et améliore la productivité lors de l'écriture et de la maintenance du code.

► 2. Console Interactive

La console interactive vous permet d'exécuter du code Python en temps réel et d'afficher les résultats instantanément. Cela facilite le processus de test et de débogage de petits morceaux de code.

► 3. Explorateur de Variables

L'explorateur de variables affiche les valeurs des variables en cours d'exécution **pendant le débogage**, ce qui vous permet de surveiller l'état de votre programme à chaque étape.

► 4. Assistant

Thonny IDE possède un assistant intégré, conçu pour **aider les débutants** à programmer en Python. L'assistant propose des suggestions contextuelles, des corrections automatiques et des explications pour guider les utilisateurs tout au long de leur processus d'apprentissage. Par exemple, lorsque vous commencez à taper du code Python, l'assistant peut suggérer des complétions de code, vous permettant ainsi d'explorer les différentes options disponibles et d'apprendre de nouvelles fonctionnalités du langage. De plus, l'assistant fournit des **informations utiles sur les erreurs de syntaxe**, ce qui peut être particulièrement bénéfique pour les étudiants en programmation qui cherchent à comprendre et à corriger leurs erreurs.

► 5. Fenêtre de Gestion des Packages

Cette fenêtre vous permet d'installer, de mettre à jour et de supprimer des packages Python supplémentaires directement depuis l'IDE.

🔔 Pour plus d'informations sur l'installation de packages, rendez-vous sur [la page "installer des modules"](#)

Comment Utiliser Thonny pour Programmer en Python

Maintenant que nous avons exploré les avantages et l'interface de Thonny, voyons comment l'utiliser pour programmer en Python :

► 1. Installation de Thonny

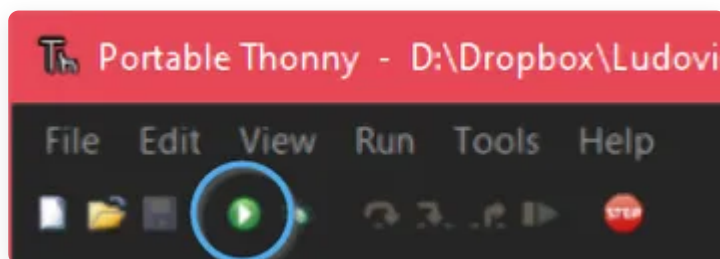
Téléchargez et installez Thonny à partir du site officiel de Thonny. Une fois l'installation terminée, lancez Thonny depuis votre menu d'applications.

► 2. Écrire du Code

Utilisez l'éditeur de code de Thonny pour écrire votre code Python. Vous pouvez commencer par des exemples simples et progresser à votre propre rythme.

► 3. Exécution du Code

Après avoir écrit votre code, vous pouvez l'exécuter en appuyant sur le bouton "Exécuter" ou en utilisant le raccourci clavier correspondant (**F5**). Les résultats s'afficheront dans la console interactive.



► 4. Débogage

Si vous rencontrez des erreurs, utilisez le débogueur intégré pour parcourir votre code pas à pas et identifier les problèmes éventuels.

► 5. Installation de Packages Additionnels

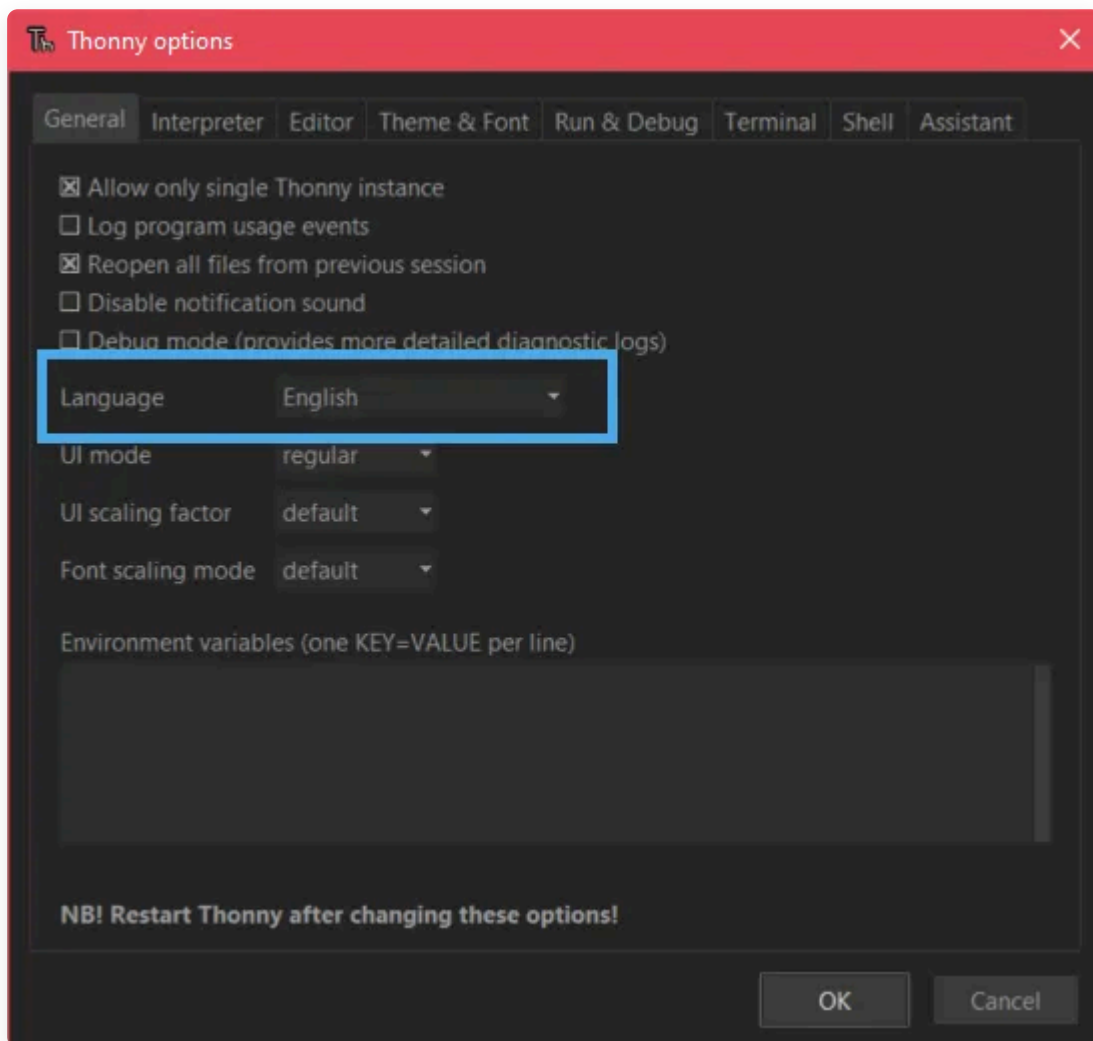
Si vous avez besoin de fonctionnalités supplémentaires, utilisez le gestionnaire de packages intégré pour installer les modules Python nécessaires à votre projet.

Configurer Thonny IDE

il est possible de configurer Thonny IDE pour qu'il corresponde à votre utilisation. Pour se faire, il faut passer par le menu "Outils (GB **Tools**)" / "Préférences". Voici les options principales.

► Changer la langue d'affichage

Pour changer de langue, rendez-vous dans l'onglet "Général" et choisissez votre langue dans la liste déroulante.



Vous devrez redémarrer Thonny pour que ce changement soit pris en compte.

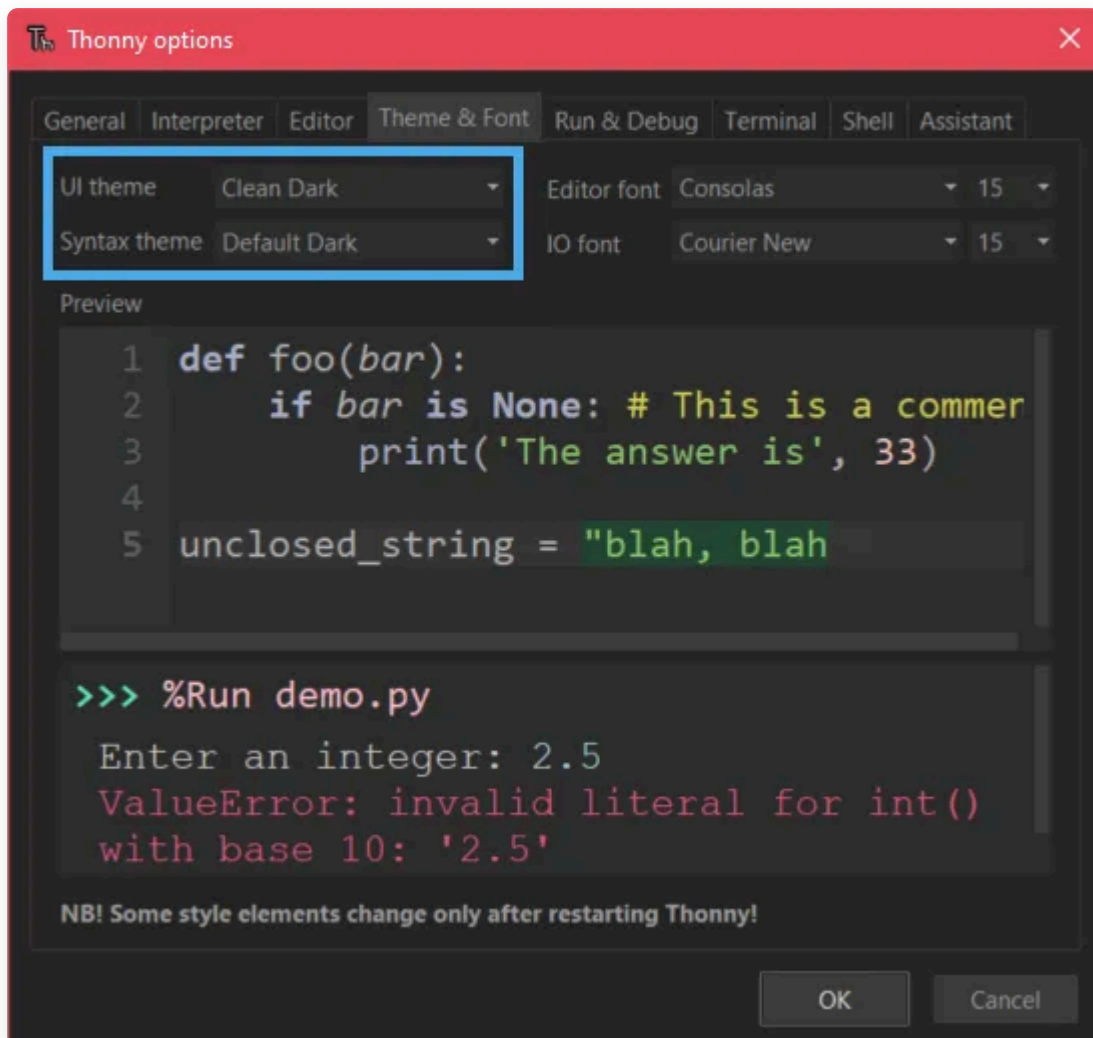
► Change le thème d'affichage

Thonny possède un **mode sombre**.

Le mode sombre, également connu sous le nom de "**dark mode**", offre un fond d'écran foncé avec un texte clair, ce qui **réduit la luminosité de l'écran** et **crée un contraste visuel agréable**. Les développeurs apprécient souvent le mode sombre pour son **aspect esthétique**, mais aussi pour ses avantages ergonomiques, notamment la **réduction**

de la fatigue oculaire lors de longues sessions de programmation, surtout dans des environnements peu éclairés.

Pour configurer le mode sombre, rendez-vous dans l'onglet "Theme & Font", et choisissez un thème pour l'interface ET pour la syntaxe (le code écrit en python):



Bien sûr, vous pouvez choisir le thème que vous convient 😊

Conclusion

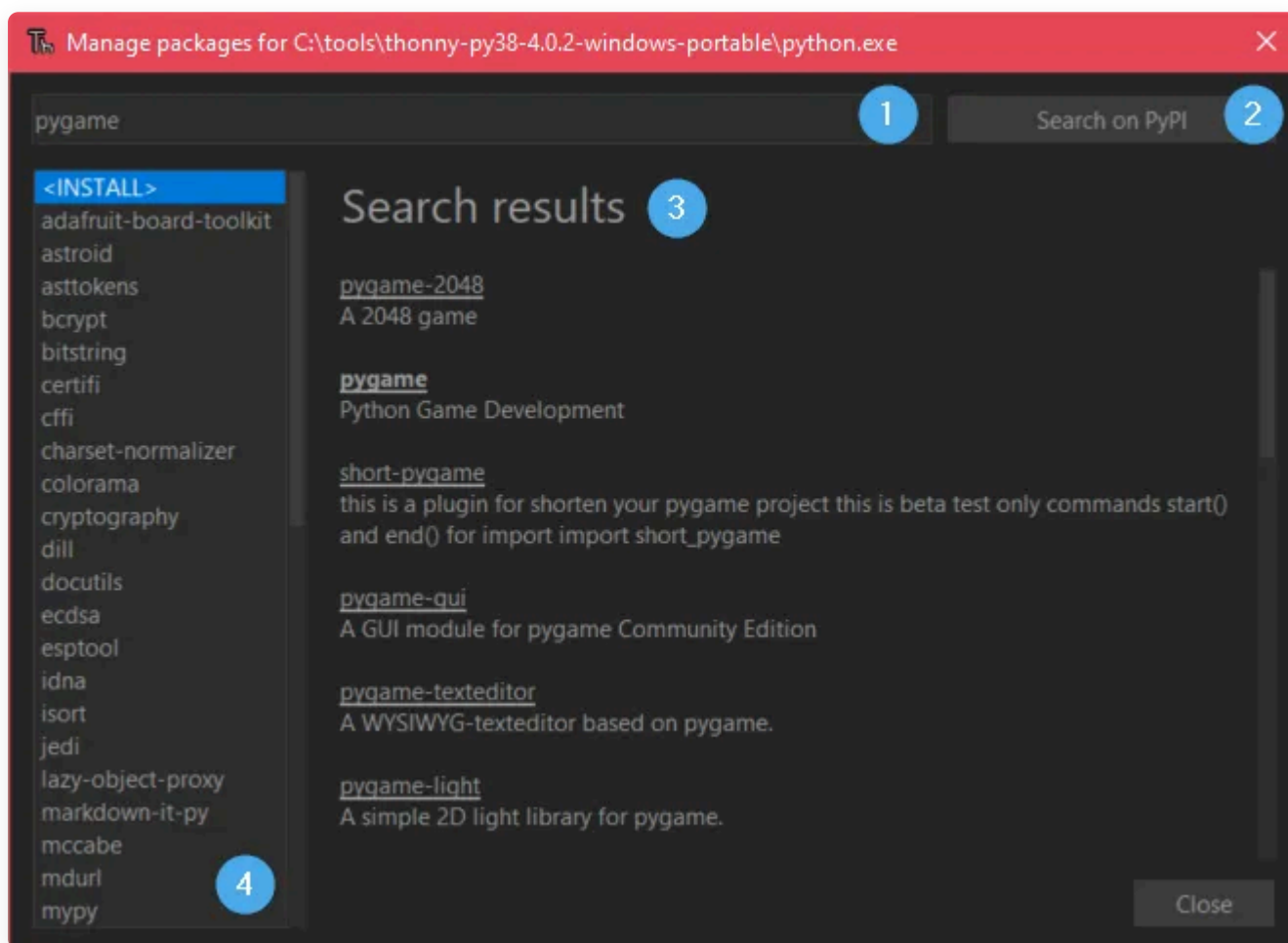
Thonny IDE est un outil exceptionnel pour les débutants en Python, offrant une expérience de développement conviviale, une interface intuitive et des fonctionnalités puissantes pour écrire, exécuter et déboguer du code Python. Que vous soyez novice en programmation ou un développeur expérimenté, Thonny peut vous accompagner tout au long de votre parcours d'apprentissage et de développement en Python. N'hésitez pas à télécharger Thonny et à commencer à explorer le monde passionnant de la programmation Python dès aujourd'hui !

Thonny Python - Installer des modules

L'une des fonctionnalités essentielles de Thonny est la capacité d'installer des modules Python supplémentaires pour étendre ses fonctionnalités. Dans ce guide, nous allons vous montrer étape par étape comment installer des modules Python dans Thonny, en incluant des captures d'écran pour une meilleure compréhension.

Accéder à la gestion des packages

Pour accéder à la gestion des packages, cliquez sur le menu **Outils** (GB **Tools**), puis sélectionnez **Gérer les packages ...** (GB **Manage packages**). Cela ouvrira une nouvelle fenêtre où vous pourrez rechercher, installer et supprimer des packages Python.

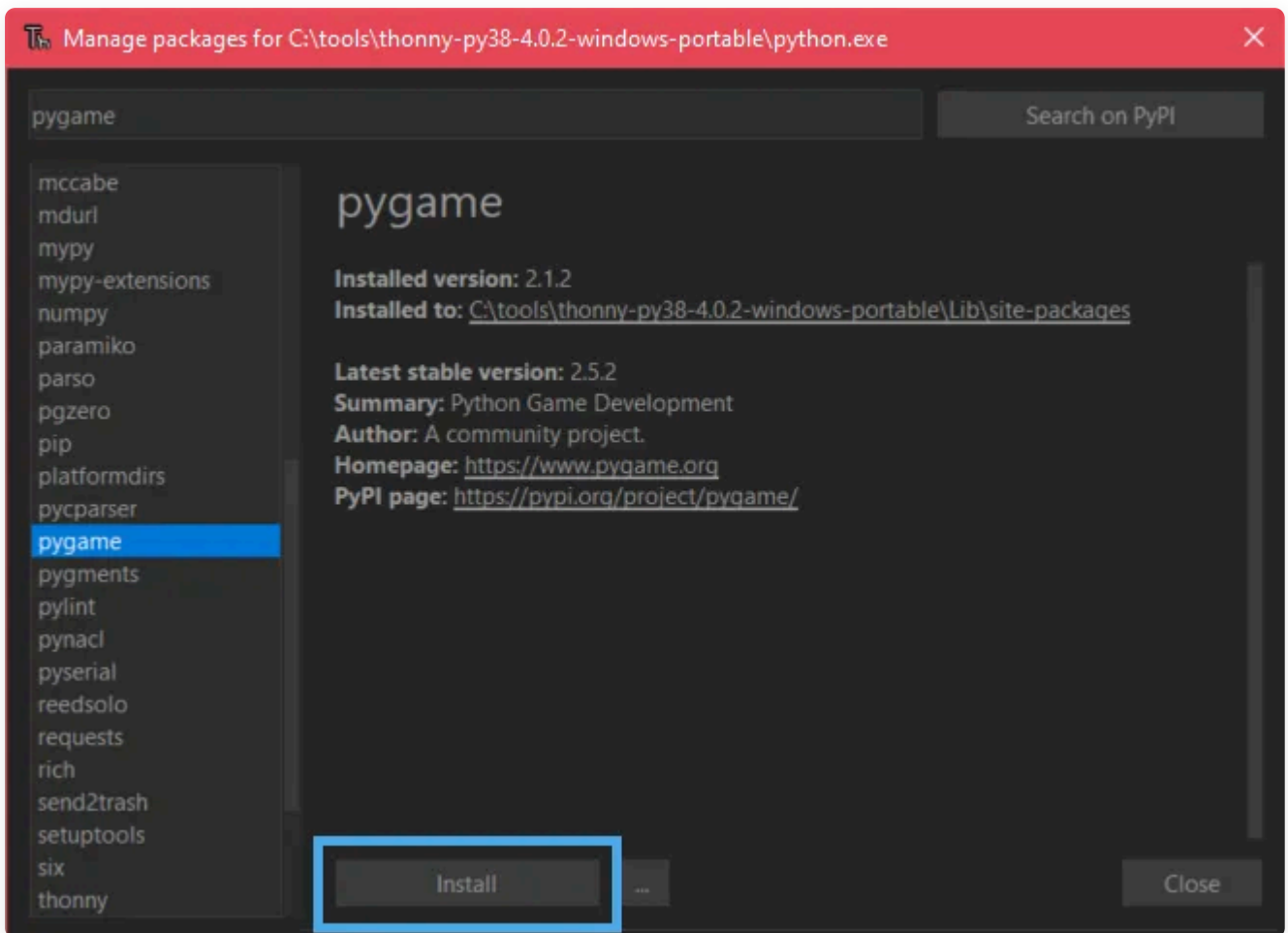


Étape 3: Rechercher et installer un module Python

Dans la fenêtre de gestion des packages, vous verrez une barre de recherche en haut [1]. Utilisez cette barre de recherche pour trouver le module Python que vous souhaitez installer. Par exemple, recherchons le module **requests**, souvent utilisé pour effectuer des requêtes HTTP dans Python. Appuyer sur [Entrée] ou cliquez sur

[2] pour valider votre recherche

Par exemple, tapez "pygame" dans la barre de recherche, puis appuyez sur la touche **Entrée** pour lancer la recherche. Une fois que le module **pygame** apparaît dans la liste des résultats, cliquez sur le nom du module [3] pour afficher la fenêtre d'installation.



Cliquez sur le bouton **Installer** pour commencer le processus d'installation.

Étape 4: Vérifier l'installation du module

Une fois l'installation terminée, vous verrez un message indiquant que le package a été installé avec succès. Vous pouvez maintenant fermer la fenêtre de gestion des packages.

Pour vérifier que le module a été correctement installé, vous pouvez l'importer dans votre script Python. Par exemple, créez un nouveau fichier Python dans Thonny et ajoutez la ligne suivante en haut du fichier :

```
import pygame
```

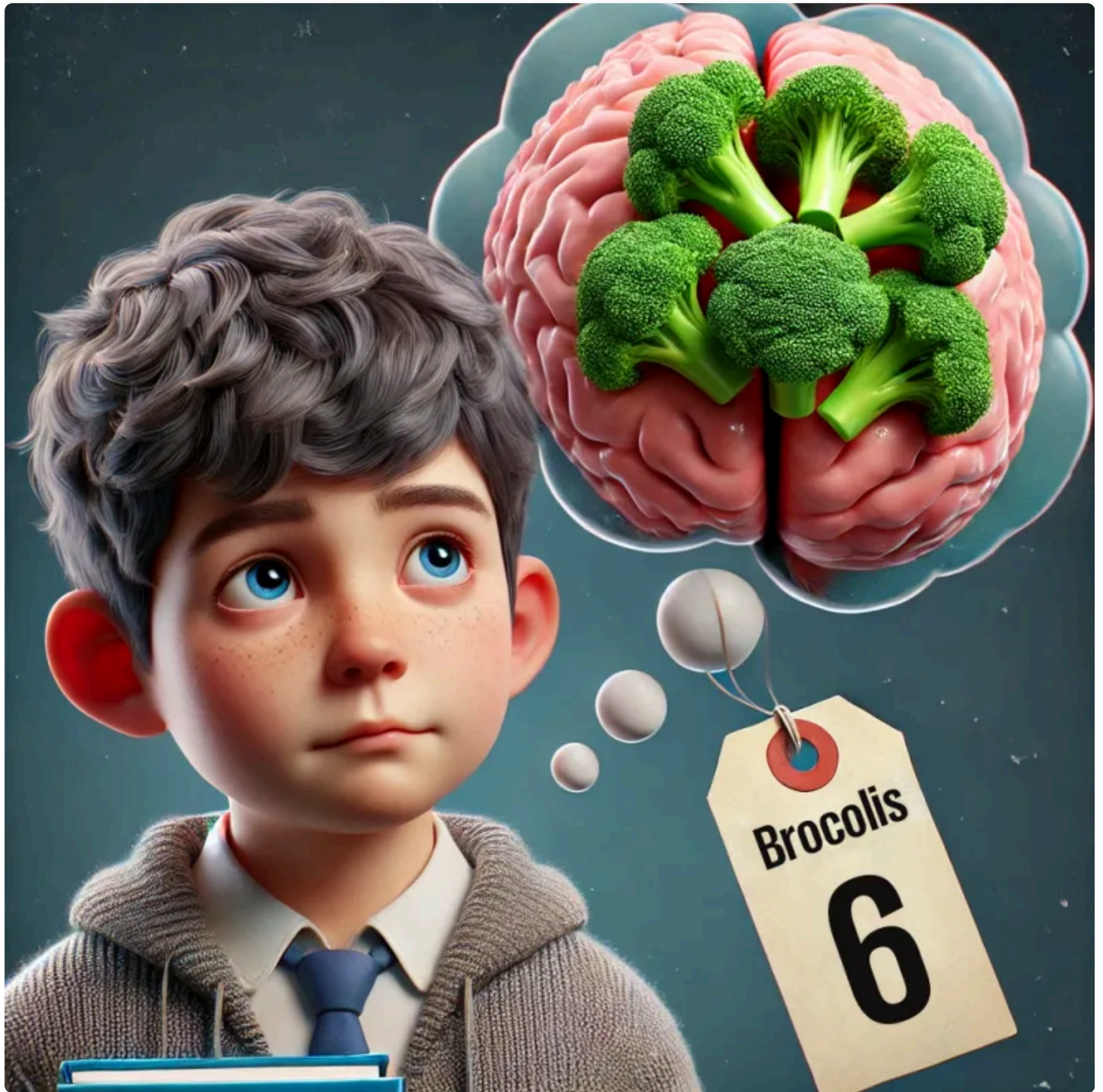
Si vous ne voyez pas d'erreurs, cela signifie que le module a été installé avec succès et que vous pouvez commencer à l'utiliser dans vos projets Python.

Conclusion

Dans ce guide, nous avons exploré comment installer des modules Python dans Thonny Python. En suivant ces étapes simples, vous pourrez étendre les fonctionnalités de Thonny et développer des projets Python plus avancés. N'hésitez pas à explorer d'autres modules disponibles et à expérimenter avec eux dans vos propres projets.

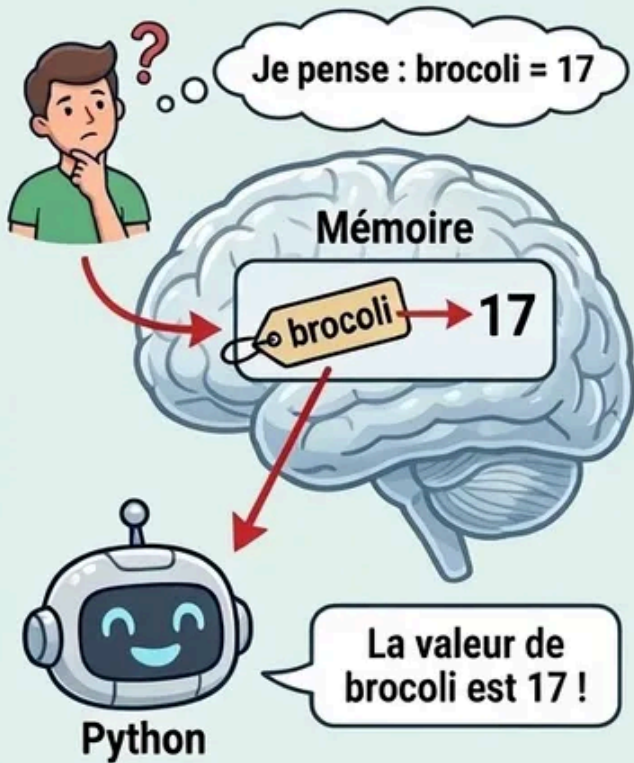
Les variables en Python

*Les variables sont l'un des concepts fondamentaux en programmation. Elles permettent de ****stocker des informations et de les utiliser dans un programme****.*



1. L'ANALOGIE CÉRÉBRALE

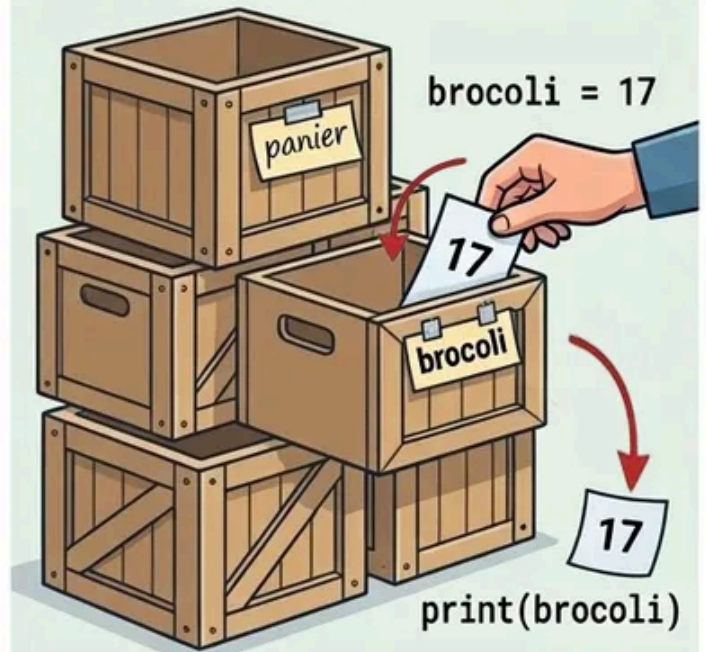
Le cerveau associe un nom à une **valeur**.



Le cerveau associe un nom à une idée.

2. L'ANALOGIE des BOÎTES (ou Caisses)

Mémoire de l'Ordinateur



Une variable est une boîte nommée pour ranger une **valeur**.

Une variable est un **espace de mémoire temporaire** où on stocke une **valeur**.

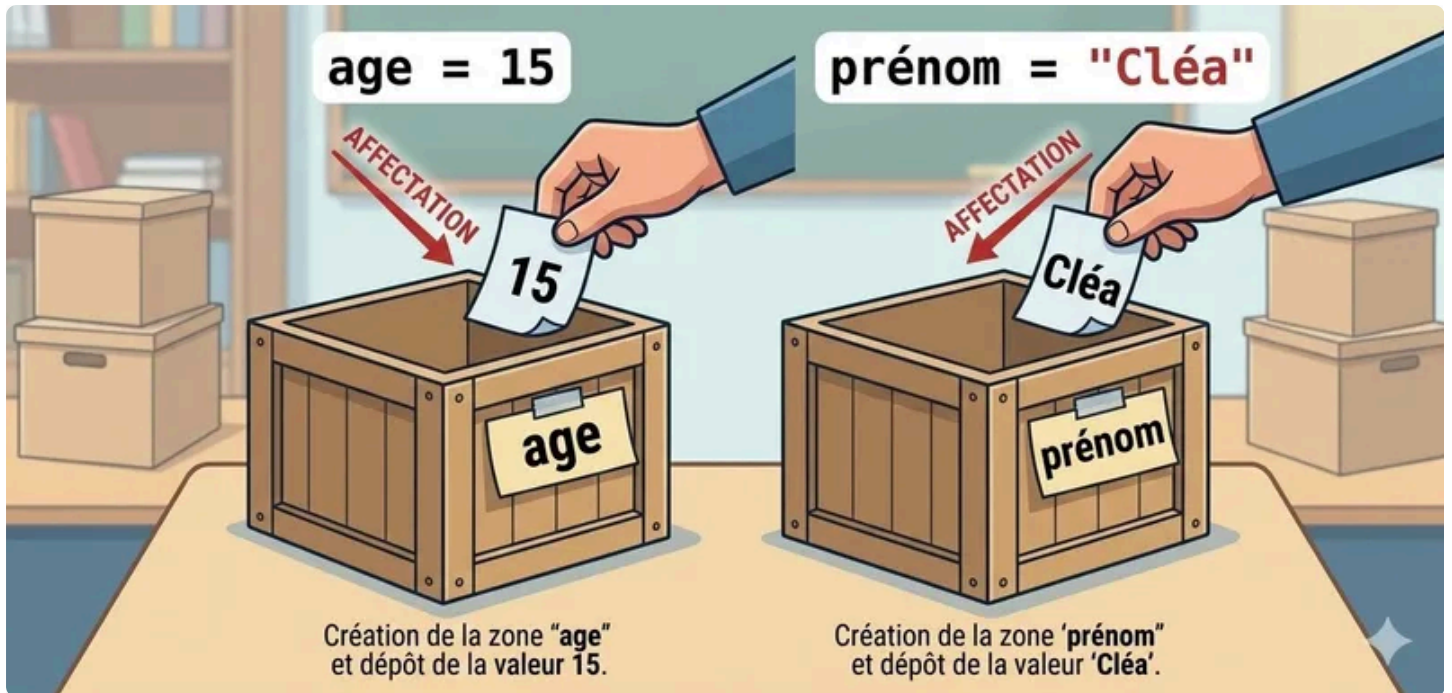
► Premier exemple

Prenons un exemple pour illustrer ce concept en Python :

```
nom = "Cléa"  
age = 15
```

Dans cet exemple :

- `nom` est une variable qui contient le texte `Cléa` ,
- `age` est une variable qui contient le nombre `15` ,

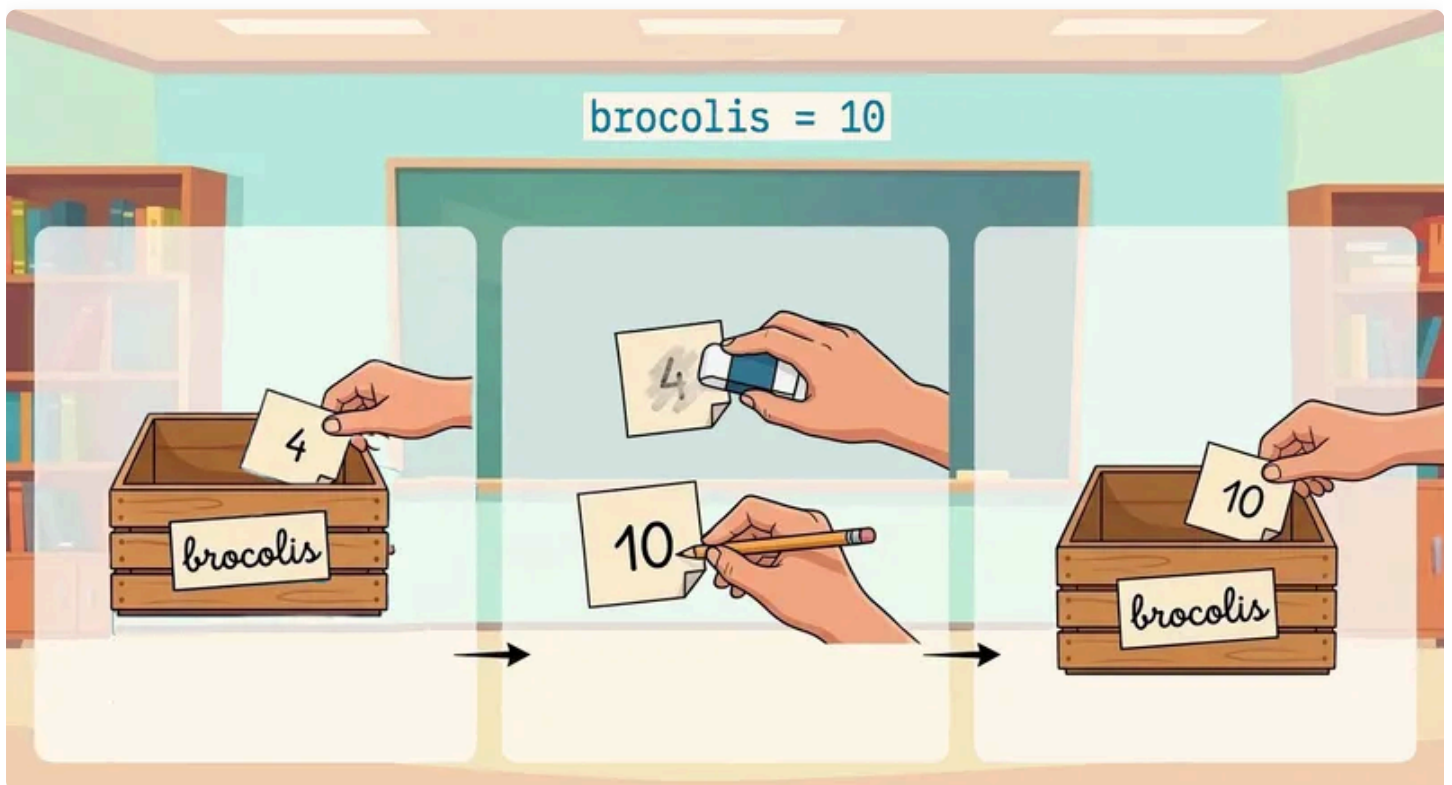


Ensuite, on pourra récupérer ces valeurs pour les afficher ou les utiliser dans un calcul par exemple.

Une variable est une zone **temporaire** dans la **mémoire** de l'ordinateur, à laquelle on donne un **nom** et dans laquelle on stocke une **valeur**.

On peut donc changer la valeur d'une variable... autant de fois qu'on veut. Imagine une variable `brocolis` qui contient `4`. On peut lui affecter une nouvelle valeur:

```
brocolis = 10
```



Importance des variables

Les variables servent à **stocker des informations** que le programme peut **utiliser** ou **modifier** au cours de son exécution. Elles agissent comme des "**boîtes**" dans lesquelles on peut ranger une valeur, par exemple un nombre, une chaîne de caractères, ou encore le résultat d'un calcul. **On utilise les variables lorsque l'on souhaite conserver une donnée pour s'en servir plus tard**, par exemple pour effectuer des **opérations**, **afficher un résultat**, ou gérer des interactions avec l'utilisateur.

Les variables permettent également de **rendre le code plus clair** et réutilisable, car elles **remplacent des valeurs fixes par des noms compréhensibles**.

En résumé, on utilise des variables pour stocker les valeurs qui seront affichées, utilisées ou modifiées plus tard dans le programme, souvent plusieurs fois.

Souviens-toi de ce schéma:



Un programme reçoit des données/informations en entrée, leur applique un traitement et fournit des données/informations en sortie. **Dans un programme informatique, il y a donc des données et des traitements.** Retiens ceci:

Les données sont stockées dans des variables.

Déclarer ses Variables

En Python, les variables sont **déclarées** en leur **affectant** une valeur.

En Python, **déclarer une variable** signifie **créer une variable** en lui donnant un **nom** et une **valeur**.

Par exemple, pour déclarer une variable nommée "age" avec une valeur de 25, nous pouvons utiliser le code suivant :

```
age = 25
```

Cela veut dire qu'on va **stocker** la valeur **25** dans la variable **nom**. Cela se fait **de droite à gauche**.

💡 Il faut **toujours** déclarer une variable avant de l'utiliser. Si tu utilises une variable avant de l'avoir déclarée, Python va afficher l'erreur **NameError**:

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'age' is not defined
>>>
```

"name 'age' is not defined" signifie que la variable `age` n'a pas été déclarée au moment où elle est utilisée.

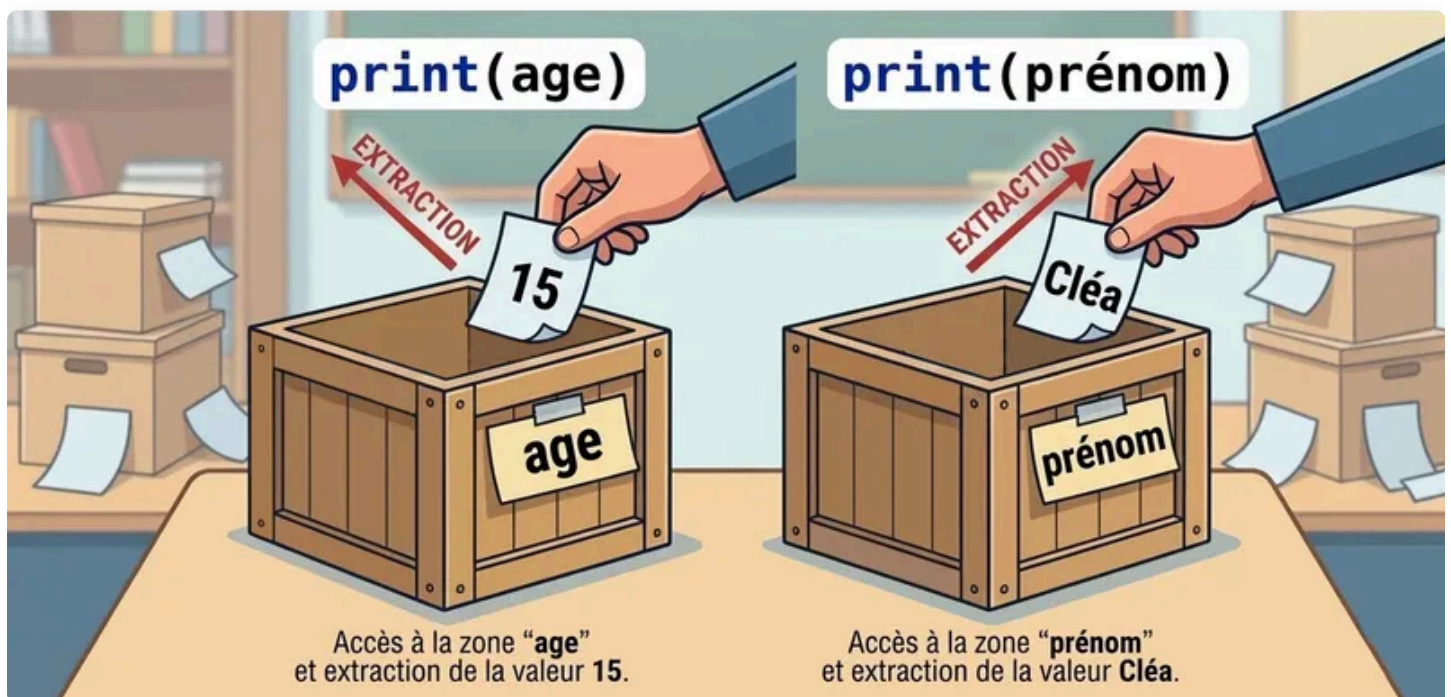
Cela peut être dû au fait que:

- tu essaies d'utiliser une variable avant de l'avoir déclarée
- il y a une faute dans le nom (ex: `prenon` au lieu de `prenom` ou une différence minuscules/majuscules)
- tu as oublié les guillemets autour d'un texte (par exemple: `print(Sara)`)

Utiliser les Variables

Une fois la variable déclarée, on peut l'afficher ou l'utiliser dans des opérations et autres instructions.

```
age = 15
prenom = "Cléa"
print (age) # Affichera 15
print (prenom) # Affichera Cléa
```

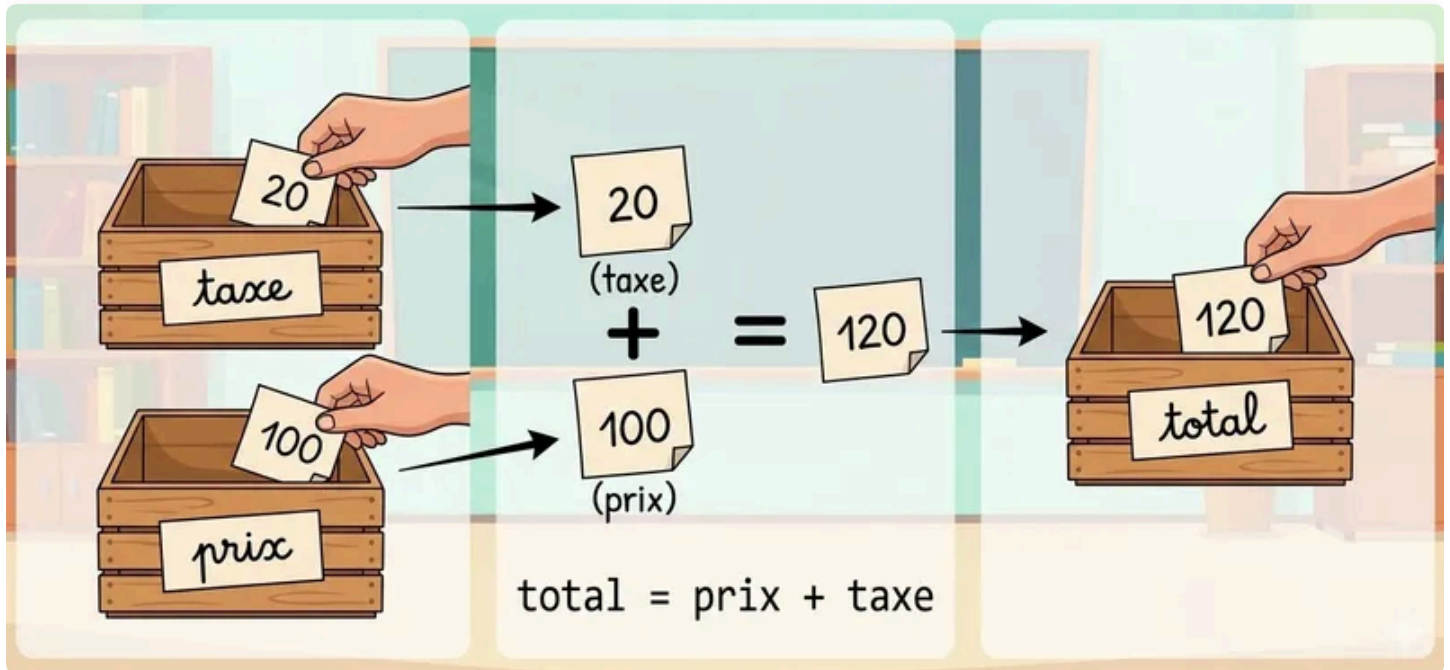


Explications

- `print` permet d'afficher quelque chose à l'écran.
- ligne 3: On affiche le **contenu** de la variable `age` : 15
- ligne 4: On affiche le **contenu** de la variable `prenom` : Cléa

```
prix = 100
taxe = 20
total = prix + taxe # total vaut maintenant 120

print (total) # Affiche 120
```



Explications

- ligne 3: On calcul le total en additionnant les **valeurs** contenues dans les variables `prix` et `taxe` et on stocke le résultat dans la variable 'total'.
- ligne 5: On affiche le **contenu** de la variable `total`, c'est à dire `120`

Ce code affiche:

```
120
```

Il est aussi possible, en une instruction, de mettre à jour la valeur d'une variable:

```
brocolis = 18

brocolis = brocolis + 4 # maintenant, brocolis vaut 22 (18+4)
```

Ligne 4: Python exécute d'abord les instructions à droite du `=`. Il récupère la valeur de `brocolis` (=18), et y ajoute 4 (=22). Python stocke ensuite le résultat (22) dans la variable `brocolis`.

Conclusion

Les variables sont une partie essentielle de la programmation. Elles permettent de **stocker et manipuler des données**. En maîtrisant les variables, tu pourras écrire des programmes plus dynamiques et puissants.

Afficher ses variables avec print

La fonction `print()` est une des bases de la programmation en Python. Elle permet d'afficher du texte, des nombres ou des résultats sur l'écran, ce qui est essentiel pour communiquer avec l'utilisateur ou vérifier le fonctionnement d'un programme.

Objectif

Apprendre à utiliser `print()` pour afficher différentes informations de manière claire et organisée sur l'écran.

Introduction à `print()`

La fonction `print()` permet d'afficher du contenu dans le terminal. Sa syntaxe de base est la suivante :

```
print(valeur)
```

(Attention aux parenthèses)

Exemple :

```
print("Bonjour tout le monde !")
```

Résultat :

```
Bonjour tout le monde !
```

Afficher plusieurs valeurs

On peut afficher plusieurs valeurs en les séparant par des virgules. Par défaut, `print()` ajoute un espace entre chaque valeur.

Exemple :

```
print("Bonjour", "tout", "le", "monde", "!")
```

Résultat :

```
Bonjour tout le monde !
```

Affichage de variables

On peut également utiliser `print()` pour afficher le contenu de variables.

Exemples :

```
annee = 2025
print(annee) # affiche: 2025
print("Nous sommes en", annee) # affiche: Nous sommes en 2025
```

```
nom = "Alice"
age = 25
print("Nom :", nom, "Âge :", age)
```

Résultat :

```
Nom : Alice Âge : 25
```

Changer le séparateur entre les valeurs

Par défaut, les valeurs affichées avec `print()` sont **séparées par un espace** (" "). On peut modifier cela grâce au paramètre `sep`.

Exemple :

```
print("Python", "est", "génial", sep="-")
```

Résultat :

```
Python-est-génial
```

Modifier la fin de ligne

Par défaut, `print()` termine chaque affichage par un saut de ligne (`\n`). On peut changer cela grâce au paramètre `end`.

Exemple :

```
print("Bonjour", end=" ")
print("tout le monde !")
```

Résultat :

```
Bonjour tout le monde !
```

À retenir

- La fonction `print()` est utilisée pour afficher du contenu.
- Elle peut afficher des chaînes, des nombres, des variables, ou des calculs.
- Les paramètres `sep` et `end` permettent de personnaliser l’affichage.

Avec une bonne maîtrise de `print()`, tu peux rendre tes programmes interactifs et clairs pour l'utilisateur ou pour toi-même !

Manipuler les variables en Python

Une variable permet de stocker une valeur temporairement. Et, surtout, elle permet à votre programme de manipuler cette valeur.

Les variables peuvent être manipulées de différentes manières en Python. Par exemple, nous pouvons leur **attribuer de nouvelles valeurs**, les **utiliser** dans des opérations mathématiques, les **concaténer** pour former des chaînes de caractères, etc.

Voici des exemples de manipulation de variables :

```
# Attribuer une nouvelle valeur à une variable
age = 26

# Utiliser des variables dans des opérations mathématiques
a = 10
b = 20
somme = a + b
difference = b - a
produit = a * b
quotient = b / a

# Concaténer des variables pour former une chaîne de caractères
prenom = "Jean"
nom = "Dupont"
nom_complet = prenom + " " + nom
```

Principes de base des manipulations

Une variable garde sa valeur jusqu'à la fin du programme ou jusqu'à ce qu'on lui en affecte une nouvelle.

Le simple fait d'utiliser une variable **ne change pas** sa valeur.

Manipuler valeurs et variables: exemples guidés

Exemples pour explorer la manipulation des variables

Petits calculs : `print(7*44)`

PLACES DANS UNE SALLE

Combien de personnes une salle de cinéma peut-elle accueillir?

```
# 7 rangées de 44 sièges  
print(7 * 44)
```

Étapes

- On a 7 rangées.
- Chaque rangée a 44 sièges.
- Total = 7×44 → on affiche le total.

GOBELETS POUR UN ÉVÉNEMENT

On achète 6 packs de 30 gobelets au magasin.

```
# 6 packs de 30 gobelets  
print(6 * 30)
```

Étapes

- 6 packs.
- 30 gobelets par pack.
- Total = 6×30 .

AUTOCOLLANTS IMPRIMÉS

On a 10 feuilles avec chacune 44 autocollants. Combien avons-nous d'autocollants?

```
# 10 feuilles de 44 autocollants  
print(10 * 44)
```

Étapes

- 10 feuilles.
- 44 autocollants par feuille.
- Total = 10×44 .

MINUTES D'ENTRAÎNEMENT

Je m'entraîne pendant 60 minutes tous les jours de la semaine. Pendant combien de minutes je m'entraîne ?

7 séances de 60 minutes

```
print(7 * 60)
```

Étapes

- 7 séances.
- 60 minutes par séance.
- Total = 7×60 minutes.

DISTANCE EN TOURS DE PISTE

7 tours de 44 mètres (petite piste)

```
print(7 * 44)
```

Étapes

- 7 tours.
- 44 m par tour.
- Distance totale = 7×44 m.

Print avec variables

PRIX TOTAL

```
prix_total = 12.5  
print(prix_total)
```

Étapes

- `prix_total` est une variable.
- Elle contient 12.5 (euros).
- `print(prix_total)` affiche la valeur.

DISTANCE

```
distance_km = 8  
print(distance_km)
```

Étapes

- `distance_km` contient 8.
- On affiche la distance.

NOMBRE DE PARTICIPANTS

Un évènement est organisé sur 4 jours. Chaque jour il peut accueillir 100 participants.

```
participants = 4 × 100
print(participants)
```

Étapes

- `participants` contient 400.
- On affiche le nombre.

Étapes

- On crée les variables
- On calcul le résultat...
- ... et on l'affiche

Assignation de valeurs et `print` (avec calcul simple et résultat)

Dans ces exercices, tu vas apprendre à **modéliser une situation réelle** avec des variables, à **effectuer un calcul simple**, puis à **afficher le résultat**. Avant d'exécuter le programme, tu dois **prédire ce que Python va afficher**, comme si tu étais l'ordinateur.

► Exercice — Calcul du prix des courses

Tu fais les courses du jour. Chaque article a un **prix**, et tu en achètes une **quantité** donnée.

Consigne

- Identifie les informations nécessaires pour calculer le prix total.
- Stocke chaque information dans une variable.
- Calcule le total à payer.
- Affiche le résultat avec `print`.

```
prix = 3
quantite = 7
total = prix * quantite
print(total)
```

► Zone de prédiction

👉 Total affiché :

► Aide à la compréhension

- `prix` représente le prix d'un article
- `quantite` représente le nombre d'articles
- `total` correspond au calcul : prix × quantité

► Exercice — Distance parcourue

Un véhicule roule à vitesse constante pendant un certain temps.

Consigne

- Stocke la vitesse et le temps dans des variables.
- Calcule la distance parcourue.
- Affiche la distance.

```
vitesse = 60
temps = 2
distance = vitesse * temps
print(distance)
```

► Zone de prédiction

👉 Distance affichée :

► Aide à la compréhension

- La distance se calcule avec : $\text{vitesse} \times \text{temps}$
- Le résultat est exprimé en kilomètres

► Exercice — Recette d'un événement

Lors d'un événement, des tickets sont vendus.

Consigne

- Stocke le prix d'un ticket.
- Stocke le nombre de tickets vendus.
- Calcule la recette totale.
- Affiche le montant.

```
prix_ticket = 11
tickets = 40
recette = prix_ticket * tickets
print(recette)
```

► Zone de prédiction

👉 Recette affichée :

► Aide à la compréhension

- Chaque ticket rapporte le même montant
- La recette est un total d'argent gagné

► Exercice – Calcul d'un salaire

Une personne travaille plusieurs heures avec un taux horaire fixe.

Consigne

- Stocke le taux horaire dans une variable.
- Stocke le nombre d'heures travaillées.
- Calcule le salaire total.
- Affiche le salaire.

```
taux = 14
heures = 5
salaire = taux * heures
print(salaire)
```

► Zone de prédiction

👉 Salaire affiché :

► Aide à la compréhension

- Le salaire dépend du nombre d'heures
- Plus on travaille, plus le salaire augmente

► Exercice – Pages imprimées

Un document est imprimé en plusieurs copies.

Consigne

- Stocke le nombre de pages d'un document.
- Stocke le nombre de copies.
- Calcule le nombre total de pages imprimées.
- Affiche le résultat.

```
pages = 18
copies = 6
total_pages = pages * copies
print(total_pages)
```

► Zone de prédiction

👉 Nombre de pages affiché :

► Aide à la compréhension

- Chaque copie contient le même nombre de pages
- Le total correspond à toutes les pages imprimées

► Message clé

Un programme commence toujours par **ce que je sais** (les variables), puis **ce que je calcule**, et enfin **ce que j'affiche**.

Changer la valeur d'une variable (la "mise à jour")

STOCK QUI DIMINUE

```
stock = 20
print(stock)

stock = 15
print(stock)
```

Étapes

- Au début : stock = 20 → on affiche 20.
- Ensuite on *remplace* la valeur : stock = 15 → on affiche 15.

DISTANCE QUI AUGMENTE (ON AJOUTE UN TRAJET)

```
distance = 5
print(distance)

distance = distance + 3
print(distance)
```

Zone de prédiction --> 👉 Valeur affichée : _____

Étapes

- Distance initiale : 5 km.
- Puis on ajoute 3 km → 8 km.
- La variable prend la nouvelle valeur.

PRIX APRÈS RÉDUCTION

```
prix = 80
print(prix)

prix = prix - 10
print(prix)
```

Zone de prédiction --> 🖱️ Valeur affichée : _____

Étapes

- Prix initial : 80.
- On applique une réduction de 10 → 70.

POINTS DANS UN JEU

```
points = 0
print(points)

points = points + 50
print(points)
```

Étapes

- Départ : 0 point.
- Gain : +50 → 50 points.

TEMPÉRATURE MISE À JOUR

```
temperature = 18
print(temperature)

temperature = 21
print(temperature)
```

Étapes

- Il faisait 18°C.
- Nouvelle mesure : 21°C.
- On affiche les deux valeurs successives.

Valeurs et variables: à toi de jouer

Dans ces exercices, tu vas apprendre à **modéliser une situation réelle** avec des variables, à **effectuer un calcul simple**, puis à **afficher le résultat**. Avant d'exécuter le programme, tu dois **prédire ce que Python va afficher**, comme si tu étais l'ordinateur.

Un programme commence toujours par **ce que je sais** (les variables), puis **ce que je calcule**, et enfin **ce que j'affiche**.

L'objectif est clair pour l'élève :  **réfléchir avant de coder**  **identifier les variables nécessaires**  **écrire le calcul**  **afficher le résultat**

Dans chaque exercice :

- identifie **les informations à connaître**
- crée **une variable par information**
- réalise **le calcul demandé**
- affiche **le résultat final**

Exercice — Courses au supermarché

Tu achètes plusieurs articles identiques.

► Consigne

- Tu connais le **prix d'un article**.
- Tu connais la **quantité achetée**.
- Calcule le **prix total à payer**.
- Affiche le total.

► Aide

 Quelles sont les **2 valeurs indispensables** pour faire le calcul ?  Quel calcul mathématique faut-il utiliser ?

Exercice — Distance parcourue en voiture

Une voiture roule à vitesse constante.

► Consigne

- Tu connais la **vitesse** (en km/h).
- Tu connais le **temps de trajet** (en heures).
- Calcule la **distance parcourue**.
- Affiche la distance.

► Aide

👉 Distance = vitesse × temps

Exercice — Temps total de sport

Tu fais plusieurs séances de sport identiques.

► Consigne

- Tu connais la **durée d'une séance** (en minutes).
 - Tu connais le **nombre de séances**.
 - Calcule le **temps total d'entraînement**.
 - Affiche le résultat.
-

Exercice — Vente de tickets

Lors d'un événement, des tickets sont vendus.

► Consigne

- Tu connais le **prix d'un ticket**.
 - Tu connais le **nombre de tickets vendus**.
 - Calcule la **recette totale**.
 - Affiche le montant gagné.
-

Exercice — Salaire journalier

Une personne travaille plusieurs heures dans la journée.

► Consigne

- Tu connais le **taux horaire**.
- Tu connais le **nombre d'heures travaillées**.
- Calcule le **salaire du jour**.

- Affiche le salaire.
-

Exercice — Pages à imprimer

Un document est distribué à toute une classe.

► Consigne

- Tu connais le **nombre de pages du document**.
 - Tu connais le **nombre d'élèves**.
 - Calcule le **nombre total de pages à imprimer**.
 - Affiche le total.
-

Exercice — Points dans un jeu

Dans un jeu, un joueur gagne le même nombre de points à chaque niveau.

► Consigne

- Tu connais le **nombre de points par niveau**.
 - Tu connais le **nombre de niveaux terminés**.
 - Calcule le **score total du joueur**.
 - Affiche le score.
-

Exercice — Consommation de carburant

Une machine consomme une quantité fixe de carburant par heure.

► Consigne

- Tu connais la **consommation par heure**.
 - Tu connais le **nombre d'heures de fonctionnement**.
 - Calcule la **consommation totale**.
 - Affiche le résultat.
-

Exercice — Boîtes de rangement

Du matériel est stocké dans des boîtes identiques.

► Consigne

- Tu connais le **nombre de boîtes**.
- Tu connais le **nombre d'objets par boîte**.
- Calcule le **nombre total d'objets stockés**.
- Affiche le total.

Exercice — Temps de visionnage

Tu regardes une série avec des épisodes de même durée.

► Consigne

- Tu connais la **durée d'un épisode**.
- Tu connais le **nombre d'épisodes regardés**.
- Calcule le **temps total de visionnage**.
- Affiche le temps total.

Voici **5 nouveaux exercices, sans code**, sur le **thème de l'astronomie**, formulés exactement dans le même esprit : réflexion → variables → calcul → affichage. Ils sont adaptés à des **débutants complets** et exploitables tels quels en classe.

Exercices – Astronomie : variables et calculs

Dans chaque exercice :

- identifie **les informations à connaître**
- crée **une variable par information**
- effectue **le calcul demandé**
- affiche **le résultat final**

Exercice — Distance parcourue par un satellite

Un satellite tourne autour de la Terre à vitesse constante.

► Consigne

- Tu connais la **vitesse du satellite** (en km par heure).
- Tu connais le **temps de déplacement** (en heures).
- Calcule la **distance parcourue** par le satellite.

- Affiche la distance.

► **Aide**

👉 Distance = vitesse × temps

Exercice — Tours de la Terre

Un satellite met toujours le même temps pour faire un tour complet de la Terre.

► **Consigne**

- Tu connais le **nombre de tours effectués**.
 - Tu connais la **durée d'un tour** (en minutes).
 - Calcule le **temps total passé en orbite**.
 - Affiche le résultat.
-

Exercice — Observations astronomiques

Un astronome observe le ciel plusieurs nuits de suite.

► **Consigne**

- Tu connais la **durée d'une observation** (en minutes).
 - Tu connais le **nombre de nuits d'observation**.
 - Calcule le **temps total d'observation**.
 - Affiche le temps total.
-

Exercice — Panneaux solaires d'un satellite

Un satellite est équipé de plusieurs panneaux solaires identiques.

► **Consigne**

- Tu connais le **nombre de panneaux solaires**.
 - Tu connais la **puissance d'un panneau**.
 - Calcule la **puissance totale produite**.
 - Affiche le résultat.
-

Exercice — Messages envoyés depuis l'espace

Une sonde spatiale envoie des données vers la Terre à intervalles réguliers.

► Consigne

- Tu connais le **nombre de messages envoyés par heure**.
- Tu connais le **nombre d'heures de communication**.
- Calcule le **nombre total de messages envoyés**.
- Affiche le total.

Ces exercices mobilisent : les variables, les opérateurs arithmétiques, `print()`, `round()`, `math`, et le formatage de chaînes — sans aucune boucle ni fonction à définir.

► Important

*En programmation, on ne commence jamais par le calcul. On commence par **identifier les informations**, puis on **les stocke dans des variables**, ensuite seulement on **calcule et on affiche**.*

Valeurs et variables: à toi de jouer (2)

Dans ces exercices, tu vas apprendre à **modéliser une situation réelle** avec des variables, à **effectuer un calcul simple**, puis à **afficher le résultat**. Avant d'exécuter le programme, tu dois **prédire ce que Python va afficher**, comme si tu étais l'ordinateur.

Un programme commence toujours par **ce que je sais** (les variables), puis **ce que je calcule**, et enfin **ce que j'affiche**.

L'objectif est clair pour l'élève :  **réfléchir avant de coder**  **identifier les variables nécessaires**  **écrire le calcul**  **afficher le résultat**

Dans chaque exercice :

- identifie **les informations à connaître**
- crée **une variable par information**
- réalise **le calcul demandé**
- affiche **le résultat final**

Voici 5 énoncés d'exercices Python à contexte scientifique, sans boucles ni fonctions :

Exercice 1 — Loi d'Ohm

Un circuit électrique contient une résistance de **470 Ω** soumise à une tension de **9 V**.

1. Calcule l'intensité du courant (en ampères) à l'aide de la loi d'Ohm : $U = R \times I$
2. Affiche le résultat avec la phrase : "L'intensité du courant est de X ampères."
3. Calcule ensuite la puissance dissipée par la résistance : $P = U \times I$ et affiche-la.

Exercice 2 — Chute libre

Un objet est lâché sans vitesse initiale depuis une hauteur de **80 mètres**.

On donne : $g = 9.81 \text{ m/s}^2$

La formule de la durée de chute est : $t = \sqrt{2h / g}$

1. Importe le module `math` et utilise `math.sqrt()` pour calculer la durée de la chute.
2. Calcule la vitesse d'impact : $v = g \times t$
3. Affiche les deux résultats en arrondissant à 2 décimales avec `round()`.



Exercice 3 – Concentration d'une solution

En chimie, la concentration molaire est donnée par : $C = n / V$

où n est la quantité de matière en moles et V le volume en litres.

On dissout **0.5 mol** de sel dans **0.25 L** d'eau.

1. Calcule la concentration molaire C .
2. On dilue ensuite cette solution en ajoutant **0.75 L** d'eau supplémentaires. Calcule la nouvelle concentration.
3. Affiche les deux concentrations avec leurs unités mol/L .



Exercice 4 – Conversion de températures

Les trois échelles de température sont liées par les formules :

- Celsius → Fahrenheit : $F = C \times 9/5 + 32$
- Celsius → Kelvin : $K = C + 273.15$

1. Stocke une température de **-40 °C** dans une variable.
2. Convertis-la en Fahrenheit et en Kelvin.
3. Affiche les trois valeurs. Que remarques-tu pour -40° ?



Exercice 5 – Distance en années-lumière

La lumière parcourt **299 792 458 m/s**. Une année-lumière est la distance parcourue par la lumière en un an.

1. Calcule le nombre de secondes dans une année (365 jours).
2. Calcule la valeur d'une année-lumière en mètres.
3. L'étoile Proxima Centauri est à **4.24 années-lumière**. Calcule sa distance en mètres et affiche-la en **notation scientifique** avec `f"{valeur:.2e}"`.



Les différentes façons d'importer

► 1. Importer le module entier

```
import math

resultat = math.sqrt(16)    # on préfixe avec "math."
print(resultat)             # 4.0
```

► 2. Importer une fonction spécifique

```
from math import sqrt

resultat = sqrt(16)      # plus besoin du préfixe "math."
print(resultat)          # 4.0
```

► 3. Importer plusieurs éléments d'un coup

```
from math import sqrt, pi, cos

print(pi)                # 3.141592653589793
print(sqrt(25))           # 5.0
print(cos(0))             # 1.0
```

► 4. Importer tout un module avec un alias (raccourci)

```
import math as m

print(m.sqrt(9))          # 3.0
print(m.pi)              # 3.141592653589793
```



Modules utiles en sciences

MODULE	CONTIENT	EXEMPLE
math	sqrt , pi , cos , sin , log , exp ...	math.pi → 3.14...
random	nombres aléatoires	random.randint(1, 6)
statistics	mean , median , stdev ...	statistics.mean([1,2,3])



Pour les exercices précédents

L'exercice 2 (chute libre) est le seul qui nécessite un import :

```
import math

h = 80
g = 9.81

t = math.sqrt(2 * h / g)
print(t)
```

Les exercices 1, 3, 4 et 5 n'ont besoin **d'aucun import** – les opérateurs `+`, `-`, `*`, `/` suffisent !

► Important

*En programmation, on ne commence jamais par le calcul. On commence par **identifier les informations**, puis on **les stocke dans des variables**, ensuite seulement on **calcule et on affiche**.*

Bien nommer ses variables en Python

Les conventions de nommage des variables en Python rendent votre code lisible et compréhensible. Elles permettent de suivre un ensemble de règles et de pratiques recommandées pour donner des noms significatifs à vos variables. Dans cet article, nous allons explorer à la fois la syntaxe obligatoire et les conventions préconisées par PEP et Google.

Syntaxe Obligatoire

En Python, il existe quelques règles de syntaxe obligatoires que vous devez respecter lors de la déclaration de vos variables :

1. **Les noms de variables doivent commencer par une lettre (a-z, A-Z) ou un trait de soulignement (_), aussi appelé "tiret du bas" ou "gb underscore").** Par exemple, `nom_variable` est valide, mais `1variable` ne l'est pas.
2. **Les noms de variables ne peuvent contenir que des lettres, des chiffres (0-9) et des traits de soulignement (_).** Les caractères spéciaux comme les espaces ou les symboles sont interdits. Par exemple, `ma_variable_1` est valide, mais `ma variable !` ne l'est pas.
3. **Les noms de variables sont sensibles à la casse.** Cela signifie que `ma_variable` et `Ma_Variable` sont considérés comme deux variables distinctes.
4. **Vous ne pouvez pas utiliser un mot réservé de Python comme nom de variable.** Par exemple, `print`, `if`, ... sont des noms de variables incorrects.

Conventions Préconisées

Il existe également des **conventions** de codage pour le langage Python pour le nommage des variables. Les conventions présentées ici sont celles **préconisées** par la communauté de développeurs Python.

Ces conventions de nommage sont **facultatives** mais **fortement recommandées**. Si chacun est libre d'utiliser ses propres conventions, utiliser des conventions "reconnues" internationalement rendront votre code plus lisible pour les autres.

Peu importe les conventions que vous utilisez, l'important est d'**être cohérent** dans votre code, votre projet et dans votre équipe. Choisissez votre convention une fois pour toutes et respectez-la.

Voici quelques-unes des conventions de nommage généralement recommandées pour les variables :

1. **Utiliser des noms de variables en minuscules avec des traits de soulignement ("tirets du bas", gb *underscores*) pour séparer les mots.** Par exemple, `ma_variable_de_test` plutôt que `MaVariableDeTest`.
- 💡 Quand on écrit un nom de variable avec des tirets du bas, on appelle ça du "snake case".
2. **Pour les constantes, utilisez des majuscules avec des traits de soulignement.** Par exemple, `VALEUR_MAXIMALE` plutôt que `valeur_maximale`.

3. **Éviter d'utiliser des caractères uniques comme noms de variables, sauf pour des variables temporaires.** Par exemple, `i` est couramment utilisé pour les indices dans les boucles, mais il est préférable d'utiliser un nom plus descriptif pour les variables importantes.
4. **Soyez descriptif dans le choix des noms de variables.** Un nom de variable comme `compteur` est plus informatif que simplement `c`.
5. De manière générale, on évite les articles dans les noms: `aire_du_rectangle` -> `aire_rectangle`

Conclusion

En conclusion, les conventions de nommage des variables en Python sont essentielles pour maintenir un **code propre et lisible**. En suivant la syntaxe obligatoire et en adoptant les conventions préconisées par la communauté, vous faciliterez la compréhension de votre code par vous-même et par les autres développeurs. N'oubliez pas que la cohérence est essentielle, alors choisissez une convention et suivez-la tout au long de votre projet.

Pratique

► Exercice : Nommer les Variables en Respectant les Conventions Python

Vous allez vous entraîner à nommer des variables en respectant les conventions de nommage en Python. Ces conventions aident à rendre votre code plus clair, lisible et maintenable.

En Python, les variables doivent être nommées en utilisant le style **snake_case**, c'est-à-dire avec des mots en minuscules séparés par des underscores `_`. Par exemple, pour "Le nombre de kilomètres parcourus", on utilise `km_parcourus`.

CONSIGNES

Pour chaque phrase ci-dessous, trouvez un nom de variable approprié en respectant les conventions Python.

1. Le nombre total d'élèves dans la classe
2. La longueur du rectangle en centimètres
3. La largeur de la salle de sport
4. Le poids de l'objet en kilogrammes
5. Le montant total des achats
6. La température de l'eau en degré Celsius
7. La vitesse maximale autorisée sur la route
8. Le nombre d'articles en stock
9. Le nom du professeur principal
10. L'âge de l'utilisateur
11. La durée du trajet en heures
12. Le niveau de batterie du téléphone
13. La date de naissance de l'utilisateur
14. Le pourcentage de réussite de l'examen
15. Le nom de la ville de résidence

16. Le nombre de pages dans le livre
17. La quantité de sucre en grammes
18. Le prix d'un billet de cinéma
19. La taille de l'écran en pouces
20. Le taux de croissance de la population
21. Le code postal de l'adresse
22. Le nombre de jours de vacances
23. La distance entre deux villes en kilomètres
24. Le numéro de la carte d'identité
25. Le niveau sonore en décibels
26. Le temps de réponse du serveur en millisecondes
27. Le prix de vente d'un produit
28. Le nombre de personnes présentes dans la salle
29. Le taux d'inflation annuel
30. La durée de vie moyenne d'une pile en heures

Exemple de réponse attendue :

- Pour "Le nombre de kilomètres parcourus", la variable est `km_parcourus` .
- Pour "La température extérieure", la variable est `temperature_exterieure` .

OBJECTIF

Entraînez-vous à bien nommer les variables de façon claire et explicite pour que d'autres développeurs puissent comprendre facilement à quoi elles servent.

N.B. : Dans cet exercice, il n'est pas nécessaire d'implémenter un programme en Python ; concentrez-vous uniquement sur le nommage des variables en respectant les conventions de style Python.

Python - Opérations mathématiques de base

Python est un langage de programmation polyvalent qui peut être utilisé pour effectuer diverses opérations mathématiques. Dans ce cours, nous allons explorer les opérations mathématiques de base en Python, notamment l'addition (+), la soustraction (-), la multiplication (), la division (/), la division entière (//), le modulo (%), et l'exponentiation (**).*

Voici un article de cours sur la syntaxe de base du langage Python pour effectuer des opérations mathématiques de base.

Pour faire court: les opérations mathématiques en Python fonctionnent **comme en math**, y compris la **priorité des opérations**.

Addition (+)

L'addition est l'opération qui permet de combiner deux nombres pour obtenir leur somme. En Python, l'opérateur d'addition est le signe "+". Voici un exemple :

```
a = 5
b = 3
resultat = a + b
print(resultat) # Affiche 8
```

Soustraction (-)

La soustraction est l'opération qui permet de trouver la différence entre deux nombres. En Python, l'opérateur de soustraction est le signe "-". Voici un exemple :

```
x = 10
y = 4
difference = x - y
print(difference) # Affiche 6
```

Multiplication (*)

La multiplication est l'opération qui permet de trouver le produit de deux nombres. En Python, l'opérateur de multiplication est le signe "*" (astérisque). Voici un exemple :

```
p = 7
q = 2
produit = p * q
print(produit) # Affiche 14
```

Division (/)

La division permet de trouver le quotient de deux nombres. En Python, l'opérateur de division est le signe "/" (slash). Voici un exemple :

```
dividende = 8
diviseur = 2
quotient = dividende / diviseur
print(quotient) # Affiche 4.0
```

Division Entière (//)

La division entière permet de trouver le quotient de deux nombres en ignorant la partie décimale. En Python, l'opérateur de division entière est le signe "//". Voici un exemple :

```
numerator = 15
denominator = 4
quotient_integer = numerator // denominator
print(quotient_integer) # Affiche 3
```

Modulo (%)

L'opérateur modulo (%) permet de trouver le reste de la division de deux nombres. Voici un exemple :

```
n = 17
m = 5
reste = n % m
print(reste) # Affiche 2
```

Exposants (**)

L'exponentiation permet d'élever un nombre à une puissance donnée. En Python, l'opérateur d'exponentiation est le double astérisque "**". Voici un exemple :

```
base = 2
exposant = 3
resultat_expo = base ** exposant
print(resultat_expo) # Affiche 8
```

Séparateur de décimales

En Belgique, nous utilisons la virgule (,) pour séparer la partie entière de la partie décimale d'un nombre réel (ce qu'on appelle un nombre à... virgule).

En Python, on utilise le point (.) pour séparer la partie entière de la partie décimale d'un nombre réel.

```
pi = 3.1415
nombre_or = 1.61803398875
```

Priorité des opérations

Les règles de **priorité** sont les **mêmes qu'en math**: les ***** et **/** sont calculés avant les **+** et les **-**.

Combien donnera le calcul suivant?

```
total = 5 + 2 * 3

print(total)
```

21? Réessaye 😊 . La réponse est 11. Etant donné l'ordre des opérations, Python calcule d'abord `2 * 3` et l'additionne ensuite à 5.

Les parenthèses permettent de changer l'ordre de priorité.

Combien donnera le calcul suivant?

```
total = (5 + 2) * 3

print(total)
```

11? Réessaye 😊 . La réponse est 21. Etant donné l'ordre des opérations, Python calcule d'abord l'intérieur des parenthèses (`5 + 2`) et multiplie le résultat par 3.

Les parenthèses à l'intérieur des parenthèses sont exécutées en premier.

L'affectation (signe " = ") est toujours effectuée en dernier. Autrement dit, Python va toujours effectuer les opérations à droite du `=` d'abord.

Conclusion

Ce cours vous a introduit aux opérations mathématiques de base en Python. Vous pouvez maintenant effectuer des opérations d'addition, de soustraction, de multiplication, de division, de division entière, de modulo, et d'exponentiation en utilisant Python. Ces concepts sont essentiels pour résoudre des problèmes mathématiques et scientifiques à l'aide de la programmation. N'hésitez pas à pratiquer ces opérations pour les maîtriser davantage.

Python: Les conditions

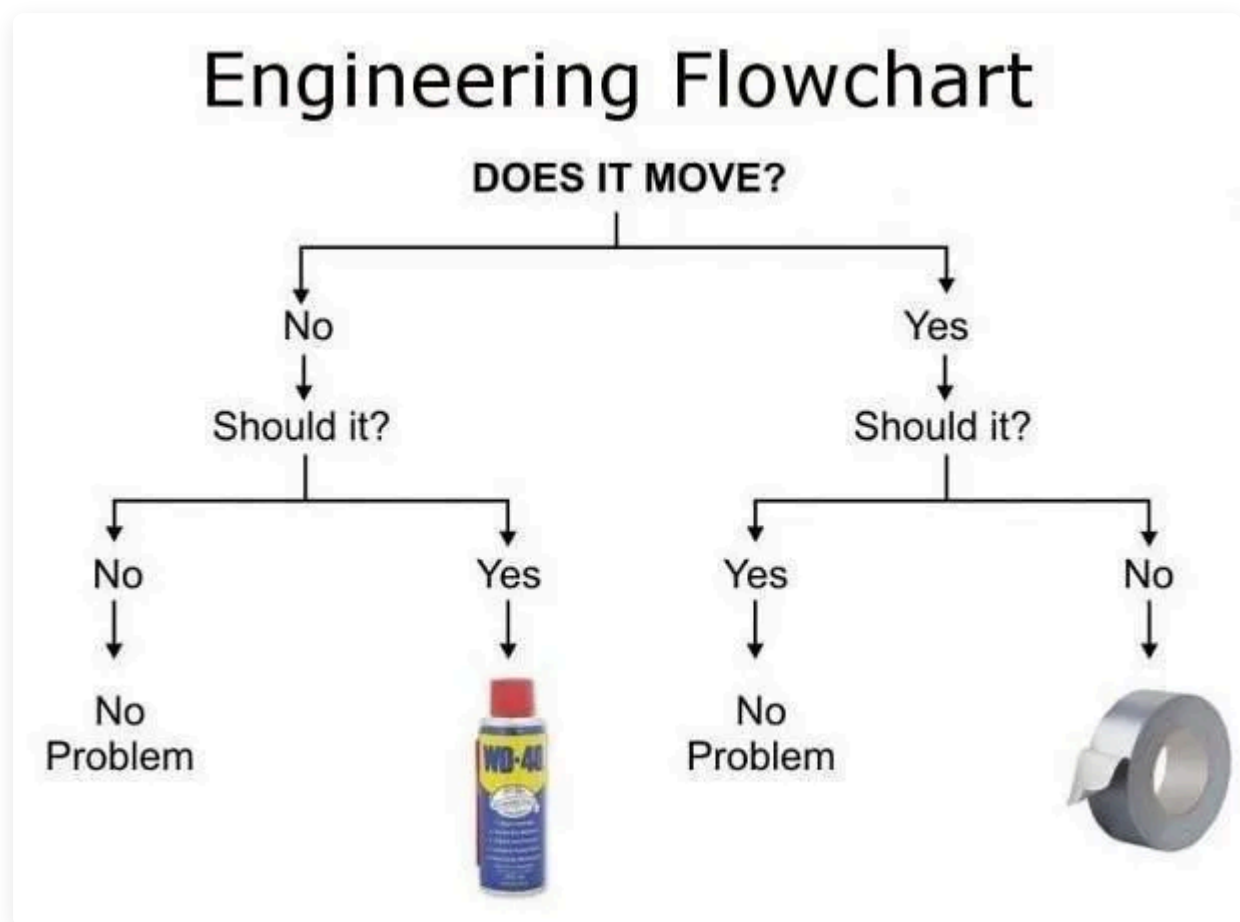
Lorsque nous débutons l'apprentissage de la programmation, l'un des concepts fondamentaux à maîtriser est celui des conditions. Elles sont au cœur de la prise de décision dans nos codes, permettant à nos programmes d'agir différemment selon les situations rencontrées. Dans cet article, nous explorerons l'importance et l'utilité des conditions en algorithmique, avec un focus particulier sur leur mise en œuvre en Python.

Les conditions sont au cœur de la prise de décision dans nos codes, permettant à nos programmes d'agir différemment selon les situations rencontrées.

Qu'est-ce qu'une condition en programmation?

Une condition en programmation est une expression qui évalue si une proposition est **vraie** ou **fausse**. Cela permet au programme de **décider** de l'exécution ou non d'un bloc de code. En Python, comme dans la plupart des langages de programmation, cela se traduit souvent par l'utilisation d'instructions `if`, `elif`, et `else` (si, sinon si, sinon).

En algorithmique, c'est ce que l'on appelle une **structure conditionnelle**.



L'utilité des conditions

Les conditions rendent nos programmes intelligents et **flexibles** et leur permettent de **prendre des décisions**.

► Adaptation aux données entrantes

Dans les applications du monde réel, les données ne sont pas statiques; elles varient constamment. Les conditions permettent à nos programmes de **s'adapter** à ces données dynamiques. Par exemple, dans une application météo, les recommandations vestimentaires peuvent changer en fonction de la température actuelle.

► Validation des entrées

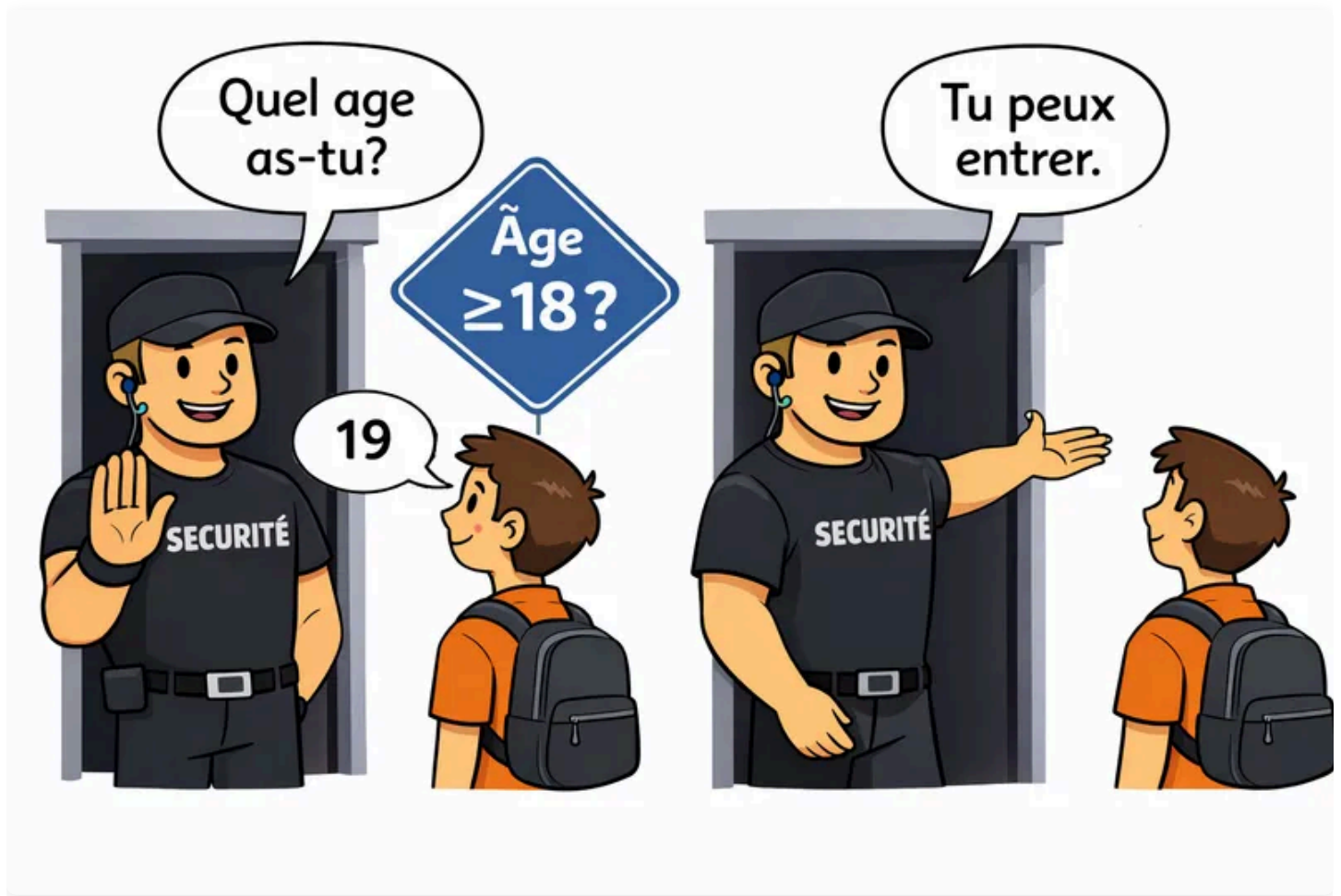
Un autre usage crucial des conditions est la **validation des entrées utilisateur**. Avant de traiter les données fournies par l'utilisateur, il est sage de vérifier leur validité. Les conditions aident à assurer que notre programme ne traite que des entrées appropriées, évitant ainsi des erreurs potentielles.

► Logique de contrôle de flux

Les conditions sont essentielles pour **contrôler le flux d'un programme**. Elles permettent de réaliser des boucles, des sauts conditionnels et des arrêts, en fonction de la logique métier de notre application. Cela rend le code non seulement plus lisible mais aussi plus efficace.

Mise en œuvre en Python

Python offre une syntaxe claire et concise pour la mise en œuvre des conditions. Voici un exemple:



```
age = 19
if age >= 18:
    print("Vous pouvez entrer.")
```

`Vous pouvez entrer.` s'affiche uniquement si `age` est plus grand que 18. Essaie avec `age = 17`

```
age = 18
if age >= 18:
    print("Vous êtes majeur.")
else:
    print("Vous êtes mineur.")
```

Dans cet exemple, le programme vérifie si l'âge de l'utilisateur est supérieur ou égal à 18. Selon le résultat, il affiche un message approprié. Python supporte également les conditions complexes avec l'utilisation des opérateurs logiques `and`, `or`, et `not`, permettant de combiner plusieurs conditions.

Conclusion

Les conditions sont un pilier de l'algorithmique et de la programmation. Elles injectent de la logique et de la flexibilité dans nos codes, permettant de construire des applications dynamiques qui répondent intelligemment à diverses situations. En maîtrisant l'usage des conditions, en particulier en Python, on ouvre la porte à la création de programmes plus complexes et plus fonctionnels, capables de résoudre des problèmes réels et d'améliorer l'interaction utilisateur.

Python: Les conditions simples

Les conditions sont un aspect fondamental de la programmation en Python. Elles permettent à votre programme de prendre des décisions en fonction de certaines conditions ou de réagir différemment en fonction de situations spécifiques. Dans cet article, nous allons explorer les structures de contrôle conditionnelles en Python, notamment les instructions `if`, `elif` (else if), et `else`.

L'instruction `if`

L'instruction `if` est utilisée pour exécuter un bloc de code si une condition spécifique est vraie (`True`). Voici un exemple :

```
age = 15
if age >= 18:
    print("Vous êtes majeur.")
```

Dans cet exemple, le code dans le bloc d'instructions sous `if` ne sera exécuté que si la condition `age ≥ 18` est vraie.

L'instruction `else`

L'instruction `else` est utilisée pour exécuter un bloc de code lorsque la condition de l'instruction `if` est fausse. Voici un exemple :

```
age = 15
if age >= 18:
    print("Vous êtes majeur.")
else:
    print("Vous êtes mineur.")
```

Dans ce cas, si la condition `age ≥ 18` est fausse, le bloc de code sous `else` sera exécuté.

L'instruction `elif` (else if)

L'instruction `elif` est utilisée pour gérer **plusieurs conditions** alternatives. Elle vous permet de tester plusieurs conditions **les unes après les autres jusqu'à ce que l'une d'entre elles soit vraie**. Voici un exemple :

```
note = 80
if note >= 90:
    print("A")
elif note >= 80:
    print("B")
elif note >= 70:
    print("C")
else:
    print("D")
```

Dans cet exemple, le code teste la note et affiche la lettre correspondante en fonction de la plage de notes.

Combinaison de Conditions

Vous pouvez également combiner des conditions en utilisant les opérateurs logiques `and`, `or`, et `not`. Voici un exemple :

```
age = 25
if age >= 18 and age <= 60:
    print("Vous êtes dans la tranche d'âge active.")
```

Cette condition vérifie à la fois si l'âge est supérieur ou égal à 18 et inférieur ou égal à 60.

Pratique

PERMIS DE CHASSE

Reprends l'exercice sur le **permis de chasse**. A la fin du programme, écris le texte " `Vous avez perdu votre permis` " si le nombre de points restants est inférieur ou égal à zéro.

DIVISION

Reprends l'exercice sur la **division entière**. Teste ton programme avec les valeurs 20 et 0. Que se passe-t-il ? Maintenant, *upgrade* ton programme de la façon suivante: Si le diviseur est différent de zéro, effectue la division, sinon, affiche le message " `Impossible de diviser par 0` ".

LES NOMBRES

Étant donné un nombre entier - imprimez "STRICTEMENT POSITIF" s'il est strictement positif. - imprimez "IMPAIR" s'il est impair et imprimez "PAIR" sinon.

IMC

Reprends l'exercice sur l'IMC. Adapte la sortie finale pour afficher l'interprétation de l'IMC selon l'OMS:

Indice de masse corporelle (IMC)

Interprétation (d'après l'OMS)

moins de 18,5	Insuffisance pondérale (maigreur)
18,5 à 25	Corpulence normale
25 à 30	Surpoids
30 à 35	Obésité modérée
35 à 40	Obésité sévère
plus de 40	Obésité morbide ou massive

Conclusion

Les structures de contrôle conditionnelles en Python vous permettent de prendre des décisions dans votre code en fonction de conditions spécifiques. Vous pouvez utiliser les instructions `if`, `elif`, et `else` pour gérer différentes situations. La maîtrise des conditions est essentielle pour écrire des programmes flexibles et réactifs. N'hésitez pas à pratiquer et à expérimenter pour approfondir vos connaissances en programmation Python.

Python: Opérations de comparaison

Les opérateurs de comparaison en Python (et dans de nombreux autres langages) vous permettent de comparer des valeurs pour évaluer des conditions et prendre des décisions dans votre code. Dans cet article, nous allons explorer les opérateurs de comparaison en Python, comment les utiliser et à quoi ils servent.

Qu'est-ce qu'un Opérateur de Comparaison?

En programmation, un opérateur de comparaison est un **symbole** ou un mot-clé qui permet de **comparer deux valeurs ou expressions**. Le résultat de cette comparaison est une valeur booléenne, c'est-à-dire **True** (vrai) ou **False** (faux). Les opérateurs de comparaison vous permettent de répondre à des questions telles que "Est-ce que ces valeurs sont **égales** ?" ou "Est-ce que cette valeur est **plus grande** que celle-là ?".

En Python, voici les principaux opérateurs de comparaison :

- `==` : Égal à
- `!=` : Différent de
- `<` : Inférieur à
- `>` : Supérieur à
- `≤` : Inférieur ou égal à
- `≥` : Supérieur ou égal à
- `... ≤ x ≤ ...` : intervalle

Exemples d'Utilisation

Voyons quelques exemples d'utilisation de ces opérateurs :

► Égal à (`==`)

```
a = 5
b = 5
resultat = (a == b) # Cela renverra True car a est égal à b.
print("Résultat:", resultat) # True
```

💡 L'opérateur de comparaison est `==` (2x le signe `=`) afin de ne pas le confondre avec l'opérateur d'affectation.

Taper ce code dans un éditeur:

```
a = 5
b = 15

print ( a == b)
print ( a = b)
```

Que se passe-t-il à la ligne 4 et à la ligne 5 ?

► Différent de (`!=`)

```
x = 10
y = 20
resultat = (x != y) # Cela renverra True car x est différent de y.
print("Résultat:", resultat) # True
```

► Inférieur à (`<`)

```
age = 15
limite_age = 18
resultat = (age < limite_age) # Cela renverra True car age est inférieur à limite_age.
print("Résultat:", resultat) # True
```

► Supérieur à (`>`)

```
note = 85
note_passage = 60
resultat = (note > note_passage) # Cela renverra True car note est supérieur à note_passage
print("Résultat:", resultat) # True
```

► Inférieur ou égal à (`<=`)

```
temps = 25
limite_temps = 30
resultat = (temps <= limite_temps) # Cela renverra True car temps est inférieur ou égal à limite_temps
print("Résultat:", resultat) # True
```

► Supérieur ou égal à (`>=`)

```
nombre1 = 50
nombre2 = 50
resultat = (nombre1 >= nombre2) # Cela renverra True car nombre1 est supérieur ou égal à nombre2
print("Résultat:", resultat) # True
```

► Intervalles

En Python, il est possible de vérifier si une valeur se trouve entre deux bornes, comme dans l'expression suivante :

```
-50 <= temperature <= 50
```

Cette syntaxe permet de vérifier en une seule étape si une valeur se trouve dans un certain intervalle. Cela revient à combiner deux comparaisons logiques à l'aide de l'opérateur logique `and`. Concrètement, l'expression ci-dessus est équivalente à :

```
-50 <= temperature and temperature <= 50
```

EXPLICATION TECHNIQUE :

1. **Lecture fluide** : L'utilisation d'une double inégalité améliore la lisibilité du code, car elle s'apparente à une notation mathématique couramment utilisée.
2. **Évaluation de gauche à droite** : Python évalue les expressions de gauche à droite. D'abord, il vérifie si `-50 ≤ temperature`. Si cette condition est vraie, il continue avec `temperature ≤ 50`. Si l'une des deux conditions échoue, l'évaluation s'arrête immédiatement.
3. **Type de valeur retournée** : Le résultat de la comparaison est un booléen (`True` ou `False`).

EXEMPLE CONCRET :

Voici un exemple qui vérifie si une température est dans une plage réaliste pour des mesures météorologiques :

```
temperature = 25

if -50 <= temperature <= 50:
    print("La température est dans la plage acceptable.")
else:
    print("La température est hors plage !")
```

Dans cet exemple, la valeur `25` répond à la condition `-50 ≤ temperature ≤ 50`, donc le programme affichera :

```
La température est dans la plage acceptable.
```

CAS D'UTILISATION :

- Vérification de plages de valeurs numériques (par exemple, température, âge, scores, etc.).
- Validation de données dans des intervalles bien définis.
- Simplification des conditions multiples pour améliorer la lisibilité du code.

BONNES PRATIQUES :

- Préférez cette notation compacte lorsque vous travaillez avec des comparaisons d'intervalles.
- Assurez-vous que cette syntaxe est claire pour votre public cible ou documentez-la si nécessaire pour éviter toute confusion.

Combinaison d'Opérateurs de Comparaison

Vous pouvez également combiner plusieurs opérateurs de comparaison pour créer des expressions plus complexes. Par exemple, vous pouvez utiliser les opérateurs `and` et `or` pour combiner des conditions.

```
age = 25
sexe = "Femme"
resultat = (age >= 18) and (sexe == "Femme") # Cela renverra True si l'âge est supérieur o
print("Résultat:", resultat) # True
```

[Lire l'article complet](#)

Pratique

Voici une série de comparaisons. Essaie d'abord de deviner le résultat de chacune d'entre elles. Tape ensuite ce code dans un éditeur pour vérifier tes hypothèses.

```
a = 5
b = 7
print(a == b)
print(a != b)
print(a > b)
print(a < b)
print(a >= b)
print(a <= b)
```

► Bonus

Quel est le résultat du code suivant?

```
a = 25
b = "25"
print(a == b)
```

[La réponse se trouve ici](#)

Conclusion

Les opérateurs de comparaison sont des outils puissants pour évaluer des conditions et prendre des décisions dans vos programmes Python. Ils vous permettent de comparer des valeurs de manière simple et efficace. En comprenant comment utiliser ces opérateurs, vous serez en mesure d'écrire un code plus précis et plus adapté à vos besoins. N'hésitez pas à les utiliser pour résoudre des problèmes et créer des applications plus sophistiquées.

Python: Conditions / Exercices de base

Voici une série d'exercices de base et cohérents pour travailler `**if / else**`, pensés pour s'exercer au début. Chaque situation a un `**sens concret**`, mobilise uniquement des `**opérations mathématiques**`, des `**input()**`, et `**une seule condition**` par exercice, sans ``and`` / ``or``, sans boucles.

Consignes générales

Envoyez vos fichiers dans le dossier `Python/Conditions simples/Exercices 01`.

► 01 – Vérifier si un nombre est positif ou négatif

`01_positif_negatif.py`

L'utilisateur encode un nombre. Afficher `"positif"` s'il est strictement supérieur à 0, sinon afficher `"négatif ou nul"`.

► 02 – Tester la majorité

`02_majorite.py`

L'utilisateur encode son âge. Afficher `"majeur"` si l'âge est au moins 18, sinon afficher `"mineur"`.

► 03 – Comparer deux nombres

`03_comparer_deux_nombres.py`

L'utilisateur encode deux nombres. Afficher `"le premier est plus grand"` si le premier est strictement supérieur au second, sinon afficher `"le second est plus grand ou égal"`.

► 04 – Pair ou impair

`04_pair_impair.py`

L'utilisateur encode un nombre entier. Afficher `"pair"` si le nombre est divisible par 2, sinon afficher `"impair"`.

► 05 – Note réussie ou ratée

`05_note_reussite.py`

L'utilisateur encode une note sur 20. Afficher `"réussi"` si la note est au moins 10, sinon afficher `"raté"`.

► 06 – Température extérieure

06_temperature.py

L'utilisateur encode une température en degrés Celsius. Afficher "il fait froid" si la température est inférieure à 10, sinon afficher "il fait bon" .

► 07 – Mot trop long ou non

07_longueur_mot.py

L'utilisateur encode un mot. Afficher "mot long" si le mot contient plus de 5 caractères, sinon afficher "mot court" .

Pour calculer la taille du mot, utiliser : `len(mot)`

► 08 – Solde bancaire suffisant

08_solde_bancaire.py

L'utilisateur encode un montant d'argent disponible. Afficher "paiement accepté" si le montant est au moins 50, sinon afficher "paiement refusé" .

► 09 – Comparaison avec zéro

09_comparaison_zero.py

L'utilisateur encode un nombre. Afficher "non nul" si le nombre est différent de 0, sinon afficher "nul" .

► 10 – Distance autorisée

10_distance_autorisee.py

L'utilisateur encode une distance en kilomètres. Afficher "trajet court" si la distance est inférieure ou égale à 10, sinon afficher "trajet long" .

► 11 – Budget suffisant

11_budget_suffisant.py

L'utilisateur encode un budget et un prix. Afficher "achat possible" si le budget est supérieur ou égal au prix, sinon afficher "achat impossible" .

► 12 – Présence d'un caractère spécial

12_caractere_special.py

L'utilisateur encode un mot. Afficher `"contient un point"` si le caractère `"."` est présent dans le mot, sinon afficher `"ne contient pas de point"`.

► 13 – Code secret

`13_code_secret.py`

L'utilisateur encode un nombre. Afficher `"code correct"` si le nombre est égal à `1234`, sinon afficher `"code incorrect"`.

► 14 – Joueur dans l'écran

`14_joueur_dans_ecran.py`

On connaît :

- `position_joueur`
- `largeur_ecran`

Afficher `"dans l'écran"` si le joueur dépasse à droite (`position_joueur > largeur_ecran`), sinon afficher `"ok"`.

(Version volontairement limitée à une seule condition, sans `and`)

► 15 – Déplacement avec retour à gauche

`15_deplacement_ecran.py`

L'utilisateur encode :

- `position_joueur`
- `largeur_ecran`
- `deplacement`

Calculer la nouvelle position. Si le joueur dépasse la largeur de l'écran, le remettre à `0`. Afficher la position finale.

► 16 – Trésor trouvé

`16_tresor.py`

On connaît `case_tresor`. L'utilisateur encode `position_joueur`. Afficher `"trésor trouvé"` si les deux valeurs sont égales, sinon `"rien ici"`.

► 17 – Points de vie après attaque

`17_points_de_vie.py`

On connaît `points_de_vie` et `attaque`. Calculer les points de vie restants. Afficher `"vivant"` si le résultat est strictement supérieur à 0, sinon `"mort"`.

► 18 – Multiples et parité

18_multiples.py

L'utilisateur encode un entier.

Afficher :

- "pair" ou "impair"
- "multiple de 2" si divisible par 2
- "multiple de 3" si divisible par 3
- "multiple de 4" si divisible par 4
- "multiple de 5" si divisible par 5
- "multiple de 6" si divisible par 6

Exemple :

```
Nombre : 12

pair
multiple de 2
multiple de 3
multiple de 4
multiple de 6
```

► 19 – Carré ou rectangle

19_carre_rectangle.py

L'utilisateur encode :

- largeur
- hauteur

Afficher "Carré" si les deux valeurs sont identiques, sinon afficher "Rectangle" .

► 20 – Achat possible ou non

20_achat_possible.py

L'utilisateur encode :

- argent_disponible
- prix_article
- quantite

Calculer le prix total de l'achat. Afficher "achat possible" si l'argent est suffisant, sinon "argent insuffisant" .

Exemple :

Argent disponible : 75
Prix d'un article : 15
Articles achetés : 4

Prix total : 100
Argent insuffisant

Python: Les conditions (combinaisons)

Dans le monde de la programmation, la capacité à combiner plusieurs conditions représente une compétence essentielle pour construire des logiciels complexes et réactifs. Cet article explore la théorie et l'utilité de cette pratique, en se concentrant sur sa mise en œuvre et ses avantages dans le développement de logiciels.

Introduction à la combinaison de conditions

La programmation conditionnelle permet aux développeurs de **diriger le flux d'exécution d'un programme** selon que certaines **conditions** sont remplies ou non. En combinant plusieurs de ces conditions, on peut créer des instructions **plus précises** et adaptatives, capables de gérer des **situations complexes** avec élégance et efficacité.

► Pourquoi combiner des conditions?

La combinaison de conditions permet de traiter des cas spécifiques qui **dépendent de plusieurs facteurs**. Imaginons une application de commerce en ligne où l'accès à certaines promotions est conditionné non seulement par le statut de membre du client mais aussi par le montant de son panier d'achat. En utilisant une combinaison de conditions, l'application peut facilement déterminer si ces critères sont remplis.

La syntaxe de base en Python

Python, avec sa syntaxe claire et intuitive, facilite la combinaison de conditions. Les opérateurs logiques tels que `and`, `or`, et `not` sont utilisés pour combiner des conditions simples en expressions plus complexes.

```
if membre == True and montant_achat >= 50:
    appliquer_promotion()
```

Dans cet exemple, la fonction `appliquer_promotion` n'est exécutée que si l'utilisateur est un membre **ET** que le montant de son achat est supérieur ou égal à 50. Cela illustre comment une combinaison de conditions peut être utilisée pour contrôler le flux d'exécution d'un programme de manière précise.

Conclusion

La combinaison de conditions est une technique puissante qui, bien maîtrisée, permet de créer des logiciels plus intelligents, réactifs et adaptés aux besoins spécifiques des utilisateurs. En suivant les bonnes pratiques de développement et en faisant preuve de rigueur dans la conception et les tests, les développeurs peuvent exploiter pleinement le potentiel de cette approche pour résoudre des problèmes complexes et offrir des expériences utilisateur enrichissantes.

Python - Les opérateurs logiques

Les opérateurs logiques permettent d'évaluer des expressions booléennes et prendre des décisions en fonction de ces évaluations. En Python, les opérateurs logiques les plus couramment utilisés sont `and`, `or`, `xor` et `not`. Dans cet article, nous explorerons chacun de ces opérateurs et leur utilisation dans des situations pratiques.

💡 Un opérateur logique permet de "combiner" 2 ou plusieurs valeurs booléennes et renvoie une valeur booléenne.

L'opérateur `and`

L'opérateur `and` retourne `True` si **les deux** expressions qu'il relie **sont vraies**, sinon il retourne `False`.

Table de vérité pour l'opérateur `and` :

A	B	A AND B
✓ True	✓ True	✓ True
✓ True	✗ False	✗ False
✗ False	✓ True	✗ False
✗ False	✗ False	✗ False

Exemple :

```
x = 5
y = 10
result = (x < 10) and (y > 5)
print(result) # Output: True
```

Dans cet exemple, l'expression `(x < 10) and (y > 5)` est vraie car les deux conditions sont satisfaites : `x` est inférieur à 10 et `y` est supérieur à 5.

Il est bien sûr possible de combiner plus de 2 comparaisons, comme dans l'exemple suivant:

Vérification de l'éligibilité pour un concours

Imaginons qu'on vérifie si une personne est éligible pour participer à un concours. Les critères sont les suivants :

- L'âge doit être supérieur ou égal à 18 ans.
- Elle doit être résidente en Belgique.
- Ca doit être sa première participation au concours.

```
age = 20
residence = "Belgique"
premiere_participation = True

if age >= 18 and residence == "Belgique" and premiere_participation:
    print("Vous êtes éligible pour participer au concours.")
else:
    print("Vous ne remplissez pas les critères pour participer au concours.")
```

Il est bien sûr possible de **combiner plusieurs comparaisons**. Dans le cas du `and`, l'expression sera considérée comme `True` si **TOUTES les conditions sont** `True`.

L'opérateur `or`

L'opérateur `or` retourne `True` si **au moins l'une** des expressions qu'il relie **est vraie**, sinon il retourne `False`.

Table de vérité pour l'opérateur `or` :

A	B	A OR B
✓ True	✓ True	✓ True
✓ True	✗ False	✓ True
✗ False	✓ True	✓ True
✗ False	✗ False	✗ False

Voici un exemple :

```
x = 5
y = 10
result = (x > 10) or (y < 20)
print(result) # Output: True
```

Dans cet exemple, l'expression `(x > 10) or (y < 20)` est vraie car au moins l'une des conditions est satisfaites : `y` est inférieur à 20.

Il est bien sûr possible de **combiner plusieurs comparaisons**. Dans le cas du `or`, l'expression sera considérée comme `True` si **au moins UNE des conditions est** `True`.

Exemple:

```
x = 5
y = 10
z = 15

if x > 0 and y < 5 and z == 15:
    print("Toutes les conditions sont remplies.")
else:
    print("Au moins une condition est fausse.")
```

L'opérateur `xor`

L'opérateur `xor` (ou exclusif) retourne `True` si une et une seule des expressions qu'il relie est vraie, sinon il retourne `False`. En Python, il n'existe pas d'opérateur `xor` natif, mais vous pouvez le simuler en utilisant l'opérateur `≠` (différent de).

Table de vérité pour l'opérateur `xor` :

A	B	A XOR B
✓ True	✓ True	✗ False
✓ True	✗ False	✓ True
✗ False	✓ True	✓ True
✗ False	✗ False	✗ False

Voici un exemple :

```
x = True
y = False
result = x != y
print(result) # Output: True
```

Dans cet exemple, l'expression `x ≠ y` est vraie car une des variables est vraie et l'autre est fausse.

L'opérateur `not`

L'opérateur `not` inverse la valeur d'une expression booléenne. Si l'expression est vraie, `not` la rendra fausse, et vice versa.

Table de vérité pour l'opérateur `not` :

A	NOT A
✓ True	✗ False
✗ False	✓ True

Voici un exemple :

```
x = True
result = not x
print(result) # Output: False
```

Dans cet exemple, la valeur de `x` est vraie, mais `not x` inverse cette valeur pour retourner `False`.

Ordre de priorité

Lorsque vous utilisez **plusieurs opérateurs logiques** dans une expression Python, il est important de comprendre leur ordre de priorité, également appelé **précédence des opérateurs**. Cela **détermine l'ordre dans lequel les opérateurs sont évalués dans une expression**. Voici l'ordre de priorité des opérateurs logiques, du plus haut au plus bas :

1. **not** : L'opérateur `not` a la plus haute priorité. Il est utilisé pour inverser la valeur d'une expression booléenne.
2. **and** : Ensuite vient l'opérateur `and`. L'expression est évaluée de gauche à droite et l'évaluation s'arrête dès qu'une valeur `False` est trouvée, car la conjonction de deux expressions nécessite que les deux soient vraies.
3. **or** : Enfin, l'opérateur `or` a la plus basse priorité. De la même manière que l'opérateur `and`, l'expression est évaluée de gauche à droite et l'évaluation s'arrête dès qu'une valeur `True` est trouvée, car la disjonction de deux expressions nécessite que au moins une soit vraie.

Il est important de noter que l'utilisation de parenthèses peut modifier l'ordre d'évaluation dans une expression. Les parenthèses ont la priorité la plus élevée et sont évaluées en premier. Par conséquent, vous pouvez utiliser des parenthèses pour spécifier explicitement l'ordre dans lequel les opérateurs doivent être évalués.

En comprenant l'ordre de priorité des opérateurs logiques, vous pouvez écrire des expressions booléennes complexes de manière claire et précise, en vous assurant que l'évaluation se déroule comme prévu.

Conclusion

Les opérateurs logiques en Python peuvent évaluer des expressions booléennes et vous aider à prendre des décisions dans vos programmes. En comprenant comment utiliser `and`, `or`, `xor` et `not`, vous pouvez écrire du code plus expressif et plus concis tout en contrôlant le flux de votre programme en fonction des conditions logiques.

Exercices

Devine le résultat de ces programmes **SANS LES EXÉCUTER** dans Python.

Ensuite, vérifie ton hypothèse en tapant le programme (PAS DE COPIER-COLLER) et en déboguant à l'aide de la touche **F7**.

1. Exercice 1 :

```
x = 5
y = 10
result = (x > 3) and (y < 20)
print(result)
```

Devinez le résultat du programme.

2. Exercice 2 :

```
a = True
b = False
c = True
result = a and b or c
print(result)
```

Devinez le résultat du programme.

3. Exercice 3 :

```
x = 15
y = 25
result = (x < 10) or (y > 20)
print(result)
```

Devinez le résultat du programme.

4. Exercice 4 :

```
a = True
b = False
result = not a or b
print(result)
```

Devinez le résultat du programme.

5. Exercice 5 :

```
x = 8
y = 12
result = (x % 2 == 0) and (y % 3 == 0)
print(result)
```

Devinez le résultat du programme.

6. Exercice 6 :

```
a = False
b = True
result = a or (not b)
print(result)
```

Devinez le résultat du programme.

7. Exercice 7 :

```
x = 20
y = 30
result = (x > 10) and (y < 25) or (x + y == 50)
print(result)
```

Devinez le résultat du programme.

8. Exercice 8 :

```
a = True
b = True
result = (not a) or (not b)
print(result)
```

Devinez le résultat du programme.

9. Exercice 9 :

```
x = 5
y = 10
z = 15
result = (x + y > 10) and (y - z < 0) or (x * z == 75)
print(result)
```

Devinez le résultat du programme.

10. Exercice 10 :

```
a = True
b = False
result = (a and b) or (not a and not b)
print(result)
```

Devinez le résultat du programme.

Ces exercices vous permettront de tester votre compréhension des opérateurs logiques en Python en essayant de prédire le résultat de chaque expression booléenne.

Python: Conditions / Combinaisons / Exercices

Exercices supplémentaires sur les conditions

Voici une série d'exercices basés sur les structures conditionnelles `if`, `else`, et `elif` et les opérateurs de combinaison `and`, `or` et `not` en Python, orientés autour du thème des jeux vidéo. Ces exercices visent à renforcer la compréhension des structures de contrôle de flux dans un contexte ludique et engageant.

Ces exercices sur le thème des jeux vidéo sont conçus pour vous aider à pratiquer la combinaison de deux ou trois conditions dans des instructions `if` à l'aide des opérateurs `OR` et `AND`.

Consignes générales

Dans les exercices suivants, quand l'utilisateur doit faire un choix parmi plusieurs propositions, vous devez

- créer une liste qui contient la liste des choix possibles
- afficher un menu de sélection Exemple:

```
1. Épée
2. Arc
3. Hache
```

Autre exemple:

```
Épée
Arc
Hache
```

- Pour le menu, choisissez la version qui correspond le mieux à ce que vous avez déjà vu en classe.
- demander son choix à l'utilisateur (`int(input( ... ))`)
- vérifier que le choix est correct. Si vous avez déjà vu les boucles, vous pouvez répéter l'étape de sélection tant que l'utilisateur effectue un choix non valide.
- Envoyez vos fichiers dans le dossier `Python/Conditions combinaisons`.

► Exercice 1: Sélection de personnage

Nom de fichier: `01.personnage.py` Demandez à l'utilisateur de choisir un personnage entre "Guerrier", "Mage" ou "Archer". Si l'utilisateur choisit "Guerrier" ou "Mage", affichez "Vous avez choisi un personnage de combat à distance."

► Exercice 2: Niveau d'accès

Nom de fichier: `02.niveau.py` Un jeu vidéo propose trois niveaux d'accès: "Débutant", "Intermédiaire", "Avancé" et "Expert". Demandez à l'utilisateur de saisir son niveau. Si l'utilisateur n'est ni "Débutant" ni "Intermédiaire", informez-le qu'il accède au niveau "Ninja".

► Exercice 3: Équipe de héros

Nom de fichier: `03.equipe.py` Dans un jeu, l'utilisateur peut choisir de rejoindre l'équipe des "Héros" ou des "Villains". Si l'utilisateur saisit "Héros" et son âge est supérieur ou égal à 18, affichez "Bienvenue dans l'équipe des Héros, adulte!".

► Exercice 4: Arme spéciale

Nom de fichier: `04.arme_speciale.py` Un joueur peut choisir une arme parmi "Épée", "Arc", ou "Bâton". Si le joueur choisit "Épée" ou "Arc" et a plus de 100 points, il obtient une version améliorée de l'arme..

► Exercice 5: Porte secrète

Nom de fichier: `05.porte_secrete.py` Dans un jeu d'aventure, pour ouvrir une porte secrète, le joueur doit posséder une clé "d'or" ou "d'argent" et connaître le mot de passe. Vérifiez si le joueur remplit les conditions.

► Exercice 6: Capacité de personnage

Nom de fichier: `06.capacite.py` Un personnage peut être "Rapide" ou "Fort". Si le personnage est "Rapide" et a moins de 50 points de santé ou s'il est "Fort" et a plus de 50 points de santé, il survit.

► Exercice 7: Accès au donjon

Nom de fichier: `07.donjon.py` Le donjon est ouvert aux joueurs de niveau 20 à 30 ou ceux qui ont un objet spécial. Demandez au joueur son niveau et s'il possède l'objet spécial (Oui/Non). Utilisez `OR` et `AND` .

► Exercice 8: Mode nocturne

Nom de fichier: `08.mode_nocturne.py` Dans un jeu, le mode nocturne est activé si le joueur est dans la "Forêt sombre" ou le "Château hanté" et qu'il est entre 22 heures et 5 heures. Utilisez `AND` et `OR` .

► Exercice 9: Compétition de jeu

Nom de fichier: `09.compétition.py` Une compétition est ouverte aux joueurs de "niveau 5" ou plus qui jouent soit à "LoL" soit à "Apex". Vérifiez l'éligibilité. Utilisez `AND` et `OR` .

► Exercice 10: Choix de quête

Nom de fichier: `10.quete.py` Un joueur doit choisir entre la quête "Chasse au trésor" ou "Sauvetage du prince". Si le joueur choisit "Chasse au trésor" et possède une carte, ou "Sauvetage du prince" et a une épée, il peut commencer. Utilisez `AND` et `OR` .

► Exercice 11: Potion magique

Nom de fichier: `11.potion_magique.py` Une potion magique est disponible si le joueur a soit des "herbes rares" soit des "cristaux magiques" et plus de 200 points de mana. Utilisez `AND` et `OR` .

► Exercice 12: Accès VIP

Nom de fichier: `12.acces_vip.py` L'accès VIP est accordé si le joueur a terminé soit le niveau "Dragon de feu" soit le "Labyrinthe des ombres" et a un score supérieur à 5000 points. Utilisez `AND` et `OR`.

Python - Les opérateurs logiques - Évaluation paresseuse

L'évaluation paresseuse, appelée aussi *short-circuit evaluation*, est une technique utilisée par les langages de programmation comme Python pour optimiser l'exécution des expressions logiques. Elle consiste à arrêter l'évaluation d'une expression logique dès qu'il n'est plus nécessaire de continuer, en fonction du résultat déjà déterminé.

Évaluation Paresseuse (Short-Circuit Evaluation)

L'évaluation paresseuse, appelée aussi **short-circuit evaluation**, est une technique utilisée par les langages de programmation comme Python pour optimiser l'exécution des expressions logiques. Elle consiste à **arrêter l'évaluation d'une expression logique dès qu'il n'est plus nécessaire de continuer**, en fonction du résultat déjà déterminé.

Principe de base

L'évaluation paresseuse repose sur les règles fondamentales des opérateurs logiques `and` et `or` :

1. Pour `and` :

- Si une condition est `False`, le résultat final sera forcément `False`, peu importe les autres conditions restantes. Il est inutile de continuer l'évaluation.
- Exemple :

```
condition1 and condition2 and condition3
```

- Si `condition1` est `False`, Python s'arrête et ignore `condition2` et `condition3`.

2. Pour `or` :

- Si une condition est `True`, le résultat final sera forcément `True`, peu importe les autres conditions restantes. Il est inutile de continuer l'évaluation.
- Exemple :

```
condition1 or condition2 or condition3
```

- Si `condition1` est `True`, Python s'arrête et ignore `condition2` et `condition3`.

Avantages de l'évaluation paresseuse

1. Amélioration des performances :

- L'évaluation paresseuse évite de calculer inutilement les conditions restantes, ce qui économise du temps et des ressources.
- Cela est particulièrement utile si certaines conditions sont complexes ou coûteuses à évaluer (exemple : fonctions ou requêtes vers une base de données).

2. Sécurité et gestion des erreurs :

- Permet d'éviter des erreurs en ne procédant pas à des évaluations inutiles.
- Exemple : Protéger une division par zéro.

```
a = 0
b = 5
if a != 0 and b / a > 1: # Python n'exécute pas b / a si a == 0
    print("Condition remplie")
else:
    print("Condition non remplie")
```

Exemples Concrets

Teste ces exemples avec Thonny en mode debug, en exécutant ces codes pas à pas (**F7**). Tu verras effectivement que toutes les conditions ne sont pas testées.

► 1. Évaluation avec `and`

```
x = 5
y = 10

if x > 0 and y / x > 1:
    print("Conditions valides")
else:
    print("Au moins une condition est invalide")
```

DÉROULEMENT :

1. Python évalue `x > 0` → Résultat : `True` .
2. Passe à `y / x > 1` → Résultat : `True` .
3. Le résultat final est donc `True` et l'instruction `print("Conditions valides")` est exécutée.

Mais si `x = 0` :

1. Python évalue `x > 0` → Résultat : `False` .
2. Python s'arrête immédiatement sans évaluer `y / x` (prévenant une division par zéro).

► 2. Évaluation avec `or`

```
def test():  
    print("Fonction exécutée")  
    return True  
  
if True or test():  
    print("Condition remplie")
```

DÉROULEMENT :

1. La première condition (`True`) est évaluée.
2. Python s'arrête immédiatement et **n'appelle pas** la fonction `test()` .

Résultat :

```
Condition remplie
```

La phrase `"Fonction exécutée"` n'apparaît pas.

► 3. Combinaison d'expressions

Un cas combinant des calculs et des protections contre des erreurs possibles.

```
a = 0  
b = 10  
  
if a != 0 and (b / a > 5 or b > 20):  
    print("Condition remplie")  
else:  
    print("Condition non remplie")
```

DÉROULEMENT :

1. Python commence par évaluer `a ≠ 0` . Si c'est `False` , il s'arrête immédiatement, sans risquer une division par zéro.

Limites et précautions

1. Effets secondaires :

- Si une condition dans une expression logique contient des fonctions qui modifient l'état d'une variable ou effectuent des actions (exemple : impression, écriture de fichiers), ces actions ne seront pas exécutées si Python n'évalue pas cette partie de l'expression.

```
def modifier_variable():  
    global x  
    x = 10  
    return True  
  
x = 0  
if False and modifier_variable(): # modifier_variable() n'est jamais exécutée  
    print("Condition remplie")  
  
print(x) # Résultat : x reste 0
```

2. Ordre des conditions :

- Placez les conditions les plus simples ou les plus probables en premier pour maximiser les avantages de l'évaluation paresseuse.

En résumé

L'évaluation paresseuse est une optimisation qui :

- Améliore les performances en arrêtant les calculs inutiles.
- Préserve la sécurité dans certains cas (comme la gestion des divisions par zéro).
- Fonctionne naturellement avec les opérateurs logiques `and` et `or`, en respectant leurs comportements.

Ce concept est particulièrement utile pour écrire du code efficace et sécurisé tout en minimisant les erreurs potentielles.

Python: Les listes

Les listes sont l'une des structures de données les plus utilisées en Python. Elles permettent de stocker une collection ordonnée d'éléments, qu'ils soient de même type ou non. Dans cet article, nous explorerons en détail ce qu'est une liste en Python, comment les créer, les manipuler et les utiliser dans vos programmes, avec des exemples concrets à chaque étape.

Qu'est-ce qu'une liste en Python ?

Une liste en Python est **une collection ordonnée d'éléments**, où chaque élément est **accessible par un index**. Les listes peuvent contenir des éléments de différents types : entiers, flottants, chaînes de caractères, voire d'autres listes ou objets plus complexes.

Création d'une liste

Pour créer une liste en Python, vous pouvez utiliser des **crochets** `[]` et séparer les éléments par des **virgules**. Voici un exemple :

```
ma_liste = [1, 2, 3, 4, 5]
```

Vous pouvez également y **ajouter des éléments** ultérieurement :

```
liste_vide = []  
liste_vide.append(1)  
liste_vide.append(2)  
liste_vide.append(3)
```

Accéder aux éléments d'une liste

Les éléments d'une liste sont accessibles par leur index, en commençant par 0 pour le premier élément. Voici comment accéder aux éléments d'une liste :

```
ma_liste = [10, 20, 30, 40, 50]  
print(ma_liste[0]) # Affiche le premier élément, soit 10  
print(ma_liste[2]) # Affiche le troisième élément, soit 30
```

Vous pouvez également utiliser des indices négatifs pour accéder aux éléments en partant de la fin de la liste :

```
print(ma_liste[-1]) # Affiche le dernier élément, soit 50
print(ma_liste[-2]) # Affiche l'avant-dernier élément, soit 40
```

► Recherche dans une liste

La recherche dans une liste consiste à vérifier si un élément est présent dans une liste donnée. Cela se fait en utilisant l'opérateur `in`. Voici un exemple :

```
liste = [1, 2, 3, 4, 5]
print(3 in liste) # Affiche True
print(6 in liste) # Affiche False
```

Modifier des éléments d'une liste

Les éléments d'une liste en Python sont **mutables**, ce qui signifie qu'ils **peuvent être modifiés** après leur création. Pour modifier un élément d'une liste, vous pouvez simplement accéder à cet élément par son index et lui assigner une nouvelle valeur :

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste[0] = 10
print(ma_liste) # Affiche [10, 2, 3, 4, 5]
```

Supprimer des éléments d'une liste

Il existe plusieurs façons de supprimer des éléments d'une liste en Python. Vous pouvez utiliser la méthode `remove()` pour supprimer un élément spécifique sur base de sa **valeur**:

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.remove(3)
print(ma_liste) # Affiche [1, 2, 4, 5]
```

Vous pouvez également utiliser le mot-clé `del` suivi de l'**index** de l'élément à supprimer :

```
del ma_liste[2]
print(ma_liste) # Affiche [1, 2, 5]
```

Ajouter des éléments à une liste

Pour ajouter un élément à une liste, vous pouvez utiliser la méthode `append()` pour ajouter l'élément à la fin de la liste :

```
ma_liste = [1, 2, 3]
ma_liste.append(4)
print(ma_liste) # Affiche [1, 2, 3, 4]
```

Exemples avec des chaînes de caractères

Les listes peuvent également contenir des chaînes de caractères. Voici un exemple :

```
liste_mots = ["Bonjour", "à", "tous"]
print(liste_mots[0]) # Affiche "Bonjour"
print(liste_mots[-1]) # Affiche "tous"
```

Vous pouvez également concaténer des listes de chaînes de caractères à l'aide de l'opérateur `+=` :

```
liste_mots = ["Bonjour", "à"]
liste_mots += ["tous"]

print(liste_mots) # Affiche ["Bonjour", "à", "tous"]
```

Conclusion

Les listes en Python sont des structures de données polyvalentes et puissantes qui permettent de stocker et de manipuler des collections d'éléments de manière efficace. Dans cet article, nous avons couvert les bases de la création, de l'accès, de la modification et de la suppression des éléments d'une liste, ainsi que l'ajout de nouveaux éléments. En comprenant ces concepts fondamentaux, vous serez en mesure d'utiliser les listes de manière efficace dans vos programmes Python.

Python: Parcourir une Liste en Python

Parcourir une liste en Python est une tâche fondamentale dans le développement de nombreux programmes. Que ce soit pour effectuer des calculs, rechercher des éléments spécifiques, ou simplement afficher le contenu, savoir comment parcourir efficacement une liste est essentiel. Dans cet article, nous explorerons les différentes méthodes pour parcourir une liste en Python, avec des exemples concrets à chaque étape.

En ce qui concerne la vidéo: 🤖😄😄😄😄😄

Parcours avec une Boucle `for`

La méthode la plus courante pour parcourir une liste en Python est d'utiliser une boucle `for`. Voici comment cela fonctionne :

```
ma_liste = [1, 2, 3, 4, 5]

for element in ma_liste:
    print(element)
```

Cette boucle `for` itère sur chaque élément de la liste `ma_liste` et affiche chaque élément à l'écran. Vous pouvez ensuite effectuer des opérations sur chaque élément à l'intérieur de la boucle.

Accès à l'Indice et à l'Élément en Parallèle

Parfois, vous avez besoin d'accéder à la fois à l'indice et à l'élément de la liste pendant le parcours. Vous pouvez le faire en utilisant la fonction `enumerate()` en combinaison avec une boucle `for`. Voici un exemple :

```
ma_liste = ['a', 'b', 'c', 'd', 'e']

for index, element in enumerate(ma_liste):
    print(f"L'élément à l'indice {index} est {element}")
```

Cette boucle `for` itère sur chaque élément de la liste `ma_liste` tout en fournissant à la fois l'indice et l'élément à chaque itération.

Parcours avec une Boucle `while`

Bien que moins courante, vous pouvez également parcourir une liste en utilisant une boucle `while` en combinant cela avec un compteur d'index. Voici un exemple :

```
ma_liste = [1, 2, 3, 4, 5]
index = 0

while index < len(ma_liste):
    print(ma_liste[index])
    index += 1
```

Cette boucle `while` utilise un index pour accéder à chaque élément de la liste `ma_liste` et les affiche à l'écran. Elle s'exécute tant que l'index est inférieur à la longueur de la liste.

Conclusion

Parcourir une liste en Python est une opération fondamentale dans de nombreux programmes. Que ce soit avec une boucle `for`, une boucle `while`, en accédant à l'indice et à l'élément en parallèle, vous avez maintenant les outils nécessaires pour parcourir efficacement des listes dans vos programmes Python. En comprenant ces différentes méthodes, vous serez en mesure de manipuler et d'exploiter les données de manière efficace dans vos projets.

Exercices

► 01 Afficher les nombres

Nom du fichier: `listes_for_01.py` Créez une liste qui contient les 9 premiers nombres premiers et écrivez un programme Python qui parcourt cette liste pour afficher chaque nombre.

```
nombres = [1, 3, 5, 7, 11, 13, 17, 19, 23]
```

Résultat attendu:

```
1
3
5
7
11
13
17
19
23
```

► 02 Jours de la semaine

Nom du fichier: `listes_for_02.py` Créez une liste contenant les jours de la semaine et affichez-la.

Résultat attendu:

```
Lundi  
Mardi  
Mercredi  
Jeudi  
Vendredi  
Samedi  
Dimanche
```

► 03 Recette de cuisine

Nom du fichier: `listes_for_03.py` Créez une liste d'ingrédients et écrivez un programme qui **affiche** chaque élément de la liste et **compte** le nombre d'éléments de cette liste (comptage en utilisant une variable intermédiaire et sans utiliser la fonction `len`).

Exemple de résultat attendu:

```
Farine  
Beurre  
Sucre  
Chocolat  
  
La liste contient 4 ingrédients.
```

► 04 Somme

Nom du fichier: `listes_for_04.py` Écrivez un programme Python qui parcourt une liste de nombres et affiche la somme de tous les nombres.

► 05 Taille

Nom du fichier: `listes_for_05.py` Écrivez un programme Python qui parcourt une liste de chaînes de caractères (noms de chiens) et **affiche la longueur** de chaque chaîne.

Exemple de résultat:

```
médor: 5  
snoopy: 6  
foulkan: 7
```

► 06 Pairs

Nom du fichier: `listes_for_06.py` Écrivez un programme Python qui parcourt une liste de nombres et affiche uniquement les nombres pairs.

Exemple de résultat:

```
2
8
12
100
250
```

► 07 Jours de la semaine

Nom du fichier: `listes_for_07.py` Créez une liste contenant les jours de la semaine et affichez-la en indiquant aussi le numéro de jour.

Résultat attendu:

```
1: Lundi
2: Mardi
3: Mercredi
4: Jeudi
5: Vendredi
6: Samedi
7: Dimanche
```

► 08 Liste de courses

Nom du fichier: `listes_for_08.py` Créez une liste de courses, affichez cette liste avec le n° de l'aliment, demandez à l'utilisateur le n° de l'élément à supprimer, supprimez l'élément et affichez la liste à nouveau.

Exemple de résultat attendu:

```
Liste:
1. Beurre
2. Pain
3. Sel
4. Salade
5. Eau

Élément à supprimer de la liste? 3

Liste:
1. Beurre
2. Pain
3. Sel
4. Eau
```

► 09 Sous-marin

Nom du fichier: `listes_for_09.py` Lorsque un sous-marin descend sous la surface de l'océan, il effectue automatiquement un balayage par sonar du fond marin à proximité. Il obtient des valeurs qui sont des mesures de la profondeur du fond marin. Ces valeurs sont enregistrées dans une liste [199, 200, 208, 210, 200, 207, 240, 269, 260, 263].

Combien de valeurs sont **supérieures** à une valeur seuil ? Demander la valeur à l'utilisateur.

Exemples de résultat attendu:

```
Mesures: [199, 200, 208, 210, 200, 207, 240, 269, 260, 263]
```

```
Seuil? 210
```

```
4 mesures sont supérieures au seuil de 210.
```

```
Mesures: [199, 200, 208, 210, 200, 207, 240, 269, 260, 263]
```

```
Seuil? 264
```

```
1 mesure est supérieure au seuil de 264.
```

```
Mesures: [199, 200, 208, 210, 200, 207, 240, 269, 260, 263]
```

```
Seuil? 270
```

```
Aucune mesure n'est supérieure au seuil de 270.
```

► 10 Sous-marin mode Kung-Fu panda

Nom du fichier: `listes_for_10.py` Lorsque un sous-marin descend sous la surface de l'océan, il effectue automatiquement un balayage par sonar du fond marin à proximité. Il obtient des valeurs qui sont des mesures de la profondeur du fond marin. Ces valeurs sont enregistrées dans une liste [199, 200, 208, 210, 200, 207, 240, 269, 260, 263].

Combien de valeurs sont **plus grandes que la valeurs précédente** ?

Résultat attendu:

```
Il y a 7 augmentations.
```

En effet :

199, 200 augmentation 200, 208 augmentation 208, 210 augmentation 210, 200 pas d'augmentation 200, 207 augmentation 207, 240 augmentation 240, 269 augmentation 269, 260 pas d'augmentation 260, 263 augmentation

► 11 Calculer la moyenne "à la main"

Nom du fichier: `listes_for_11.py` Étant donnée une liste de valeurs, calculer la moyenne des éléments de cette liste à l'aide d'une boucle `for`.

Exemple de résultat attendu:

```
Liste: [4, 8, 16, 32, 64]
Moyenne: 24.8
```

► 12 Élément du milieu

Nom du fichier: `listes_for_12.py` Étant donnée une liste de valeurs, affichez l'élément qui se trouve **exactement** au milieu de la liste. Pour cela, la liste doit contenir un nombre impair de valeurs. Si ce n'est pas le cas, affichez le message " `Pas de milieu... pour l'instant ;)` ".

Exemples de résultats attendus:

```
Liste: ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"]
Milieu: Jeudi
```

```
Liste: [1,2,3,4,5,6]
Pas de milieu... pour l'instant ;)
```

Indice: pour trouver le milieu, utilisez `//`.

► 13 ÉlémentS du milieu

Nom du fichier: `listes_for_13.py` Étant donnée une liste de valeurs, affichez le.s élément.s qui se trouve.nt au milieu de la liste.

Si la liste contient un nombre impair de valeurs, affichez la valeur du milieu, sinon affichez les éléments qui se trouvent de part et d'autre du milieu.

Exemples de résultats attendus:

```
Liste: ["Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Août", "Septembre",
"Octobre", "Novembre", "Décembre"]
Milieu: Juin / Juillet
```

```
Liste: [1,2,3,4,5,6]
Milieu: 3 / 4
```

Indice: pour trouver le milieu, utilisez `//`.

► 14 Calculer la médiane d'une liste de valeurs

Nom du fichier: `listes_for_14.py` La médiane d'une liste de valeurs est la valeur qui divise la liste en deux parties égales lorsque les valeurs sont triées dans l'ordre croissant ou décroissant. Si la liste a un nombre impair de valeurs, la médiane est simplement la valeur au milieu de la liste. Si la liste a un nombre pair de valeurs, la médiane est la moyenne des deux valeurs du milieu.

Instructions :

1. Ecrivez une fonction nommée `calculer_mediane` qui prend en entrée une liste de nombres et retourne la médiane de la liste.

2. Si la liste a un nombre **impair** de valeurs, la fonction doit retourner la valeur au milieu de la liste après avoir trié les valeurs.
3. Si la liste a un nombre **pair** de valeurs, la fonction doit retourner la moyenne des deux valeurs du milieu après avoir trié les valeurs.
4. Testez votre fonction avec différentes listes de valeurs pour vous assurer qu'elle calcule correctement la médiane dans tous les cas possibles.

Rappel : Pour trier une liste de valeurs en Python, vous pouvez utiliser la méthode `sort()` ou la fonction `sorted()`. La méthode `sort()` modifie la liste directement, tandis que la fonction `sorted()` retourne une nouvelle liste triée sans modifier l'originale.

Exemple de résultat attendu:

```
Liste: [99,86,87,88,111,86,103,87,94,78,77,85,86]
Médiane: 87
```

Python: Exercices sur les listes

Voici quelques exercices pour apprendre à utiliser les listes.

Exercices

Voici 10 énoncés d'exercices pour apprendre à créer et manipuler des listes simples sans utiliser de boucles. Ces exercices sont conçus pour se concentrer sur la création, la modification, et l'accès aux éléments des listes.

► Exercice 1 : Créer une liste de fruits

Nom du fichier: `listes_1.py` Crée une liste nommée `fruits` contenant les éléments suivants : `"pomme"`, `"banane"`, `"orange"`. Affiche la liste entière à l'aide de `print`.

► Exercice 2 : Accéder à un élément d'une liste

Nom du fichier: `listes_2.py` En utilisant la liste `fruits` que tu as créée dans l'exercice précédent, affiche uniquement le deuxième élément de la liste.

► Exercice 3 : Modifier un élément de la liste

Nom du fichier: `listes_3.py` Toujours avec la liste `fruits`, remplace l'élément `"banane"` par `"kiwi"`. Affiche la liste modifiée.

► Exercice 4 : Ajouter un élément à une liste

Nom du fichier: `listes_4.py` Crée une liste vide appelée `nombres`. Ajoute-y successivement les nombres `5`, `10`, et `15` en utilisant la méthode `.append()`. Affiche la liste à chaque ajout.

► Exercice 5 : Longueur d'une liste

Nom du fichier: `listes_5.py` Utilise la liste `nombres` de l'exercice précédent et affiche le nombre d'éléments qu'elle contient à l'aide de la fonction `len()`.

► Exercice 6 : Dernier élément

Nom du fichier: `listes_6.py` Affiche le dernier élément de la liste fruit.

► Exercice 7 : Supprimer un élément d'une liste

Nom du fichier: `listes_7.py` Crée une liste `couleurs` contenant `"rouge"`, `"vert"`, `"bleu"`, `"jaune"`. Supprime l'élément `"bleu"` en utilisant la méthode `.remove()` et affiche la liste modifiée.

► Exercice 8 : Vérifier la présence d'un élément dans une liste

Nom du fichier: `listes_8.py` Crée une liste `animaux` contenant `"chien"`, `"chat"`, `"oiseau"`. Utilise l'opérateur `in` pour vérifier si `"chat"` est dans la liste et affiche un message comme `"Le chat est dans la liste"`.

Python: Les boucles

*Les boucles sont des structures de contrôle fondamentales en programmation, permettant de **répéter l'exécution d'un bloc de code** un certain nombre de fois ou tant qu'une condition spécifiée reste vraie. Elles sont largement utilisées dans tous les langages de programmation pour automatiser des tâches répétitives, traiter des ensembles de données et réaliser des opérations itératives sur des structures complexes.*

Les boucles sont largement utilisées dans tous les langages de programmation **pour automatiser des tâches répétitives** et **traiter des ensembles de données**.

Utilité des Boucles en Programmation

En programmation, tu as souvent besoin de répéter une action plusieurs fois. Sans boucle, tu dois copier-coller le même code plusieurs fois, ce qui rend la maintenance difficile.

► Compter sans boucle

Pour afficher les nombres de 1 à 5, tu écris :

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Chaque instruction est rédigée individuellement.

► Compter avec une boucle

Utilise une boucle `for` pour automatiser la répétition :

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

La boucle parcourt automatiquement les nombres de 1 à 5. Tu écris le code une seule fois, ce qui simplifie la modification et améliore la lisibilité.

Les avantages des boucles

1. **Gain de temps** : Moins de code à écrire.
2. **Facilité de modification** : Modifier la plage de répétition est simple.
3. **Lisibilité améliorée** : Le code est plus structuré et compréhensible.

4. **Réduction des erreurs** : Moins de répétitions manuelles réduit le risque d'erreurs.

Utilité des boucles

► Répétition de tâches

Les boucles permettent d'exécuter un bloc de code plusieurs fois sans avoir à le répéter manuellement. Cela permet d'économiser du temps et des efforts en automatisant des tâches répétitives.

► Traitement de données

Les boucles sont couramment utilisées pour parcourir des ensembles de données, tels que des listes, des tableaux ou des fichiers, et effectuer des opérations sur chaque élément de manière itérative. Cela permet de traiter efficacement de grandes quantités de données.

Les Types de Boucles en Programmation

Il existe principalement deux types de boucles en programmation :

► 1. Boucles `for`

Les boucles `for` sont utilisées pour parcourir une séquence de données, telle qu'une liste, un tuple ou un dictionnaire, et exécuter un bloc de code pour chaque élément de la séquence. Elles sont couramment utilisées lorsque le nombre d'itérations est connu à l'avance ou lorsqu'il est nécessaire de parcourir une séquence de données.

► 2. Boucles `while`

Les boucles `while` sont utilisées pour exécuter un bloc de code tant qu'une condition spécifiée reste vraie. Elles sont couramment utilisées lorsque le nombre d'itérations n'est pas connu à l'avance ou lorsqu'une condition spécifique doit être vérifiée avant chaque itération.

Utilisation des Boucles en Python

En Python, les boucles sont largement utilisées pour diverses tâches de programmation en raison de leur simplicité et de leur flexibilité. Python offre les boucles `for` et `while`, ainsi que des fonctions intégrées telles que `range()` pour faciliter la création de boucles complexes.

À retenir

- Une boucle permet de répéter des instructions de manière automatique.
- La syntaxe de base en Python pour une boucle `for` est `for variable in range(début, fin)`.
- Utilise les boucles pour rendre ton code plus efficace et facile à maintenir.

En conclusion, les boucles sont des outils essentiels en programmation, offrant un moyen efficace de **répéter des tâches**, de traiter des données et de contrôler le flux d'exécution d'un programme.

Python: Les boucles for

Quand un programme commence à devoir répéter une action plusieurs fois, copier-coller la même ligne devient vite absurde. La boucle `for` est la réponse propre à ce problème : elle décrit une fois ce qu'il faut faire, et Python s'occupe de le refaire autant de fois qu'il faut.

Tu vas voir ici **pourquoi** on a inventé les boucles `for`, **comment** elles se construisent, et tu termineras en écrivant ta première table de multiplication automatique.

Objectifs

À la fin de ce cours, tu seras capable de :

1. Reconnaître les situations où une boucle `for` remplace utilement du copier-coller.
2. Écrire une boucle `for` qui exécute une action **un nombre précis de fois**.
3. Utiliser `range()` avec un, deux ou trois paramètres.
4. **Parcourir** une chaîne de caractères, lettre par lettre.
5. Résoudre seul un petit problème classique (table de multiplication) avec une boucle.

Partie 1 — Le problème : copier-coller, ça ne tient pas

Imagine qu'on te demande d'afficher 5 fois **le même message** — par exemple « Joyeux anniversaire, Léa ! » pour faire une carte un peu folle. Sans boucle, ça donne :

```
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
```

Cinq lignes identiques. Ça marche, mais maintenant :

- **Et s'il en fallait 200 fois ?** Tu vas vraiment écrire 200 `print` ?
- **Et si on veut changer le texte** (mettre « Bon anniversaire » à la place) ? Il faudrait modifier les 200 lignes.
- **Et si on ne sait qu'au moment de l'exécution combien de fois répéter** (l'utilisateur entre le nombre) ? Tu ne peux même pas écrire les `print` à l'avance.

 **La répétition manuelle ne passe pas à l'échelle.** Dès qu'on commence à copier-coller plus de deux ou trois lignes presque identiques, c'est le signe qu'il faut sortir une boucle.

ÉLÉMENT	RÔLE
<code>for</code>	Mot-clé qui démarre la boucle
<code>i</code>	Variable qui prend une nouvelle valeur à chaque tour
<code>in</code>	Mot-clé qui sépare la variable de la séquence
<code>range(5)</code>	Génère la séquence <code>0, 1, 2, 3, 4</code>
<code>:</code>	Termine l'en-tête (obligatoire en Python)
Corps indenté	Le code à répéter à chaque tour

⚠ **L'indentation est obligatoire.** Le corps de la boucle DOIT être indenté (4 espaces, ou une tabulation, mais reste cohérent). Sans indentation, Python ne sait pas où finit la boucle.

Partie 3 – Anatomie d'une exécution

Reprends ce code, mais cette fois on **utilise** la variable `i` :

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

Résultat :

```
Tour numéro 0
Tour numéro 1
Tour numéro 2
Tour numéro 3
Tour numéro 4
```

À chaque tour, `i` prend une valeur différente :

TOUR	VALEUR DE <code>i</code>	LIGNE AFFICHÉE
1	0	<code>Tour numéro 0</code>
2	1	<code>Tour numéro 1</code>
3	2	<code>Tour numéro 2</code>
4	3	<code>Tour numéro 3</code>
5	4	<code>Tour numéro 4</code>

💡 **Pourquoi ça commence à 0 ?** Parce que `range(5)` produit `0, 1, 2, 3, 4` – cinq valeurs en partant de 0. C'est une convention en programmation : on compte à partir de 0. On verra dans la partie 5 comment forcer un autre point de départ.

Partie 4 – Parcourir une chaîne de caractères

Il existe une autre situation où une boucle `for` est très utile : quand on veut **traiter chaque élément** d'une suite, sans forcément savoir combien il y en a.

Une chaîne de caractères, c'est exactement ça : une suite de lettres.

```
mot = "Python"

for lettre in mot:
    print(lettre)
```

Résultat :

```
P
y
t
h
o
n
```

Ici, on n'a pas écrit `range(6)`. On a écrit `for lettre in mot`. Python sait **tout seul** parcourir la chaîne lettre par lettre, peu importe sa longueur. Si le mot fait 6 lettres, la boucle s'exécute 6 fois. S'il en fait 50, elle s'exécute 50 fois.

🧠 **À retenir :** `for` ne sert pas seulement à compter. Sa vraie nature est de dire « pour chaque élément de cette suite, fais ceci ». Une chaîne, une liste, un fichier ouvert... tout ça se parcourt avec `for`.

► Exemple – Lister les lettres d'un mot

Tu pourrais essayer ça :

```
mot = "ornithorynque"

for lettre in mot:
    print("Une lettre :", lettre)
```

Sans rien savoir d'autre, ton programme va afficher 13 lignes. Tu n'as pas eu à écrire « 13 » dans ton code – c'est Python qui s'adapte.

Partie 5 — La fonction `range()` en détail

Tu as vu `range(5)`. Mais `range()` peut prendre **un, deux ou trois** paramètres. Chaque forme est utile dans des contextes différents.

► Forme 1 — `range(fin)` : de 0 à fin - 1

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

→ 0, 1, 2, 3, 4

► Forme 2 — `range(début, fin)` : de début à fin - 1

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

→ 1, 2, 3, 4, 5

💡 **Très pratique** pour compter de 1 à 10 « comme un humain », sans commencer à 0.

► Forme 3 — `range(début, fin, pas)` : avec un incrément personnalisé

```
for i in range(0, 20, 2):  
    print(i)
```

→ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18

Le pas peut être **négatif** pour aller à reculons :

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

→ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 (parfait pour un compte à rebours)

► Récapitulatif

FORME	GÉNÈRE	EXEMPLE TYPIQUE
<code>range(n)</code>	0, 1, ..., n - 1	Répéter n fois
<code>range(a, b)</code>	a, a+1, ..., b - 1	Compter de 1 à 10
<code>range(a, b, pas)</code>	a, a+pas, ... (jusqu'à < b)	Pairs, impairs, à rebours

⚠ **La valeur de fin n'est jamais incluse.** `range(1, 11)` va de 1 à 10, pas jusqu'à 11. Ça surprend au début, mais c'est ce qui permet d'écrire `range(len(mot))` sans erreur.

Exercices

► Exercice guidé — La table de 7

Énoncé. Écris un programme qui affiche la **table de multiplication par 7**, de 7×1 jusqu'à 7×10 , dans ce format :

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
...
7 x 10 = 70
```

Avant de coder, prends 30 secondes pour réfléchir. **Ne lis pas la suite tout de suite** — essaie de réfléchir à ces trois questions :

1. Combien de lignes vais-je afficher au total ?
2. Qu'est-ce qui **change** d'une ligne à l'autre ?
3. Qu'est-ce qui **reste pareil** ?

ÉTAPE 1 — IDENTIFIER LA SÉQUENCE

Dix lignes à afficher : pour `i` qui va de **1 à 10**. C'est exactement ce que produit `range(1, 11)` (rappel : la valeur de fin n'est pas incluse, donc 11 pour aller jusqu'à 10).

Premier brouillon, juste pour vérifier que la boucle tourne bien :

```
for i in range(1, 11):
    print(i)
```

Exécute. Tu dois voir afficher les nombres de 1 à 10. Si oui, **la boucle est bonne** — il ne reste plus qu'à changer ce qu'on affiche.

ÉTAPE 2 — CALCULER LE PRODUIT

À chaque tour, on veut afficher `7 x i`. En Python, le `x` s'écrit `*` :

```
for i in range(1, 11):
    print(7 * i)
```

Tu obtiens :

```
7
14
21
28
35
42
49
56
63
70
```

Les bons résultats sont là. Mais le format n'est pas celui demandé.

ÉTAPE 3 – COMPOSER LA LIGNE COMPLÈTE

On veut afficher `7 x 1 = 7`, `7 x 2 = 14`, etc. La fonction `print` accepte plusieurs arguments séparés par des virgules : elle les affiche tous séparés par un espace.

```
for i in range(1, 11):
    print(7, "x", i, "=", 7 * i)
```

Résultat :

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

C'est exactement ce qui était demandé.

ÉTAPE 4 (BONUS) – PLUS PROPRE AVEC UNE F-STRING

Quand la ligne à afficher devient longue, les `print(a, "x", b, ...)` deviennent illisibles. Python a une syntaxe plus élégante : les **f-strings** (chaînes formatées).

```
for i in range(1, 11):
    print(f"7 x {i} = {7 * i}")
```

Le préfixe `f` devant les guillemets active la magie : tout ce qui est entre `{ }` est évalué et inséré dans la chaîne. C'est exactement la même sortie qu'avant, mais beaucoup plus lisible.

 **Habitue-toi aux f-strings dès maintenant.** C'est la façon moderne d'afficher du texte mêlé à des variables en Python. Tu vas en croiser dans presque tous les exercices à partir d'ici.

POUR ALLER PLUS LOIN

Quelques variantes à tester par toi-même :

- **Table choisie** : remplace `7` par une variable `n = 7` en début de programme. Tu pourras changer la table en modifiant une seule ligne.
- **Table interactive** : remplace `n = 7` par `n = int(input("Quelle table ? "))`. Le programme demande à l'utilisateur quelle table afficher.
- **Toutes les tables de 1 à 10** : imbrique deux boucles `for` (mais ça, c'est pour un cours suivant).

► Exercice 2 — Prédiction

Avant de lancer chacun de ces programmes, **prévois sur papier** ce qu'il va afficher. Puis vérifie.

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

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

```
for i in range(1, 11, 2):  
    print(i)
```

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

► Exercice 3 — Initiales en chaîne

Écris un programme qui demande ton prénom à l'utilisateur, puis affiche chaque lettre en majuscule, **une par ligne**. Indice : la méthode `.upper()` sur une lettre la met en majuscule.

```
Ton prénom ? Ludovic  
L  
U  
D  
O  
V  
I  
C
```

📌 À retenir

- Une boucle `for` répète un bloc de code, **soit un nombre précis de fois, soit pour chaque élément d'une séquence**.
- La syntaxe générale est :

```
for variable in sequence:  
    instructions
```

- `range(n)` produit `0, 1, ..., n - 1`. `range(a, b)` part de `a`. `range(a, b, pas)` ajoute un incrément (positif ou négatif).
- La **valeur de fin n'est jamais incluse** : `range(1, 11)` s'arrête à 10.
- Une chaîne de caractères est **itérable** : `for lettre in mot` parcourt les lettres une par une.
- L'indentation du corps de la boucle est **obligatoire**.
- Les **f-strings** (`f"... {variable} ..."`) sont la façon moderne d'insérer des variables dans du texte.

Suite

Tu sais maintenant *répéter*. La prochaine étape : faire **évoluer une variable** d'un tour à l'autre. Ça permet de **compter** (combien de voyelles dans un mot ?) et **cumuler** (quelle est la somme des notes ?). C'est ce qu'on découvre dans le prochain cours.

Python: Les boucles for (exercices)

Voici des *exercices progressifs*, directement alignés avec l'article sur les boucles `for`. Ils vont du *très guidé* vers le *raisonnement autonome*, sans introduire de notions non vues.

Ces exercices te permettent de t'entraîner à utiliser les boucles `for` pour répéter des actions et parcourir des séquences (`range` , chaînes de caractères).

► 01 – Dire bonjour plusieurs fois

`01_bonjour_5_fois.py`

Afficher le mot `"Bonjour"` 5 fois à l'aide d'une boucle `for` .

► 02 – Compter de 0 à 4

`02_compter_0_a_4.py`

Afficher les nombres de `0` à `4` , chacun sur une ligne.

► 03 – Compter de 1 à 10

`03_compter_1_a_10.py`

Afficher les nombres de `1` à `10` inclus, chacun sur une ligne.

► 04 – Afficher les nombres pairs

`04_nombres_pairs_0_20.py`

Afficher tous les nombres pairs entre `0` et `20` (inclus).

► 05 – Table de multiplication de 5

`05_table_5.py`

Afficher la table de multiplication de `5` (de `5 × 1` à `5 × 10`).

► 06 – Table de multiplication (nombre de l'utilisateur)

`06_table_nombre_utilisateur.py`

Demander un nombre à l'utilisateur et afficher sa table de multiplication (de `0` à `10`).

► 07 – Carrés des nombres

07_carres_1_a_10.py

Afficher le carré des nombres de 1 à 10.

Exemple attendu :

```
1 → 1
2 → 4
3 → 9
```

► 08 – Parcourir un mot

08_parcourir_python.py

À partir du mot "Python", afficher chaque lettre sur une ligne différente.

► 09 – Parcourir un mot donné par l'utilisateur

09_parcourir_mot_utilisateur.py

Demander un mot à l'utilisateur et afficher chaque lettre sur une nouvelle ligne.

► 10 – Compter les lettres d'un mot

10_compter_lettres_mot.py

Demander un mot à l'utilisateur et afficher le **nombre total de lettres**, en comptant avec une boucle `for` (sans utiliser `len()`).

► 11 – Afficher une phrase caractère par caractère

11_phrase_caractere_par_caractere.py

À partir de la phrase "Je code en Python", afficher chaque caractère (y compris les espaces), un par ligne.

► 12 – Compter les lettres "a"

12_compter_a.py

Demander un mot à l'utilisateur et compter combien de fois la lettre "a" apparaît.

► 13 – Somme des nombres

13_somme_1_a_100.py

Calculer la somme des nombres de 1 à 100 à l'aide d'une boucle `for`.

► 14 – Afficher les positions d'un mot

14_positions_programmation.py

Afficher les nombres de 0 jusqu'à la longueur du mot "Programmation" moins 1.

► 15 – Mot vertical numéroté

15_mot_vertical_numerote.py

Demander un mot à l'utilisateur et afficher chaque lettre précédée de son numéro de position (à partir de 1).

Exemple :

```
1 : P
2 : y
3 : t
4 : h
5 : o
6 : n
```

► 16 – Exercice bonus : Le jeu du 21 (version for)

16_bonus_jeu_21_for.py

Le joueur commence avec un score de 0 . Le jeu se déroule en 5 tours maximum à l'aide d'une boucle for .

À chaque tour :

- un nombre aléatoire entre 1 et 6 est généré ;
- ce nombre est ajouté au score ;
- le tirage et le score actuel sont affichés.

À la fin du jeu :

- si le score est **exactement 21** → afficher "Bravo, 21 !"
- si le score est **inférieur à 21** → afficher "Pas assez"
- si le score est **supérieur à 21** → afficher "Trop élevé"

Contraintes :

- utiliser une boucle for
- utiliser une variable de cumul score
- pas de boucle while
- pas de break

Python: Compter et cumuler des valeurs dans une boucle

Lorsqu'on utilise une boucle, on ne fait pas que répéter des instructions. Très souvent, on veut aussi **compter des éléments** ou **additionner des valeurs** au fil des répétitions. Cet article explique comment fonctionnent ces mécanismes, comment les coder correctement, et surtout comment **raisonner** pour éviter les erreurs.

Pourquoi compter ou cumuler dans une boucle ?

Une boucle `for` exécute un bloc de code **plusieurs fois**. Il est donc naturel de vouloir :

- **compter** combien de fois quelque chose se produit (ex. : nombre de lettres, nombre de valeurs, nombre d'essais) ;
- **cumuler** des valeurs (ex. : somme de nombres, total de points, score final).

👉 Ces deux idées reposent sur **le même principe fondamental**.

Le principe clé : une variable qui évolue

Pour compter ou additionner, on utilise toujours une **variable qui change de valeur** à chaque itération.

Cette variable doit :

1. être **initialisée avant la boucle** ;
2. être **modifiée à chaque tour de boucle**.

Compter : augmenter de 1 à chaque fois

► Exemple simple : compter jusqu'à 5

```
count = 0

for i in range(5):
    count = count + 1

print(count)
```

► Ce qui se passe réellement

- `count` commence à `0`
- À chaque passage dans la boucle, on ajoute `1`
- Après 5 passages, `count` vaut `5`

👉 La ligne importante est :

```
count = count + 1
```

On peut la lire comme :

« le nouveau `count` devient l'ancien `count` plus 1 »

Cumuler : additionner une valeur qui change

► Exemple : somme des nombres de 1 à 5

```
total = 0

for i in range(1, 6):
    total = total + i

print(total)
```

► Interprétation

- `total` commence à `0`
- `i` prend successivement les valeurs `1`, `2`, `3`, `4`, `5`
- À chaque tour, on ajoute `i` au total

👉 Ici, la ligne clé est :

```
total = total + i
```

Cela signifie :

« j'ajoute la valeur actuelle de `i` au total existant »

Différence entre compter et cumuler

OBJECTIF	VARIABLE MODIFIÉE	CE QU'ON AJOUTE
Compter	<code>count</code>	toujours <code>+1</code>
Cumuler	<code>total</code>	une valeur (<code>i</code> , un score, un prix, etc.)

Erreur fréquente à éviter ❌

► Réinitialiser la variable dans la boucle

```
for i in range(5):  
    count = 0      # ❌ erreur  
    count = count + 1
```

➡ Ici, `count` est remis à `0` à chaque tour. Résultat final : `count` vaut toujours `1`.

👉 L'initialisation doit toujours être faite avant la boucle.

Astuce essentielle : raisonner sur papier ✎

Quand un programme devient difficile à suivre, **il faut le simuler**.

► Méthode recommandée

1. Trace une **ligne par variable**
2. Fais évoluer les valeurs **itération par itération**

► Exemple de simulation sur papier

Code :

```
total = 0  
  
for i in range(1, 4):  
    total = total + i
```

Tableau de suivi :

TOUR	I	TOTAL AVANT	TOTAL APRÈS
1	1	0	1
2	2	1	3
3	3	3	6

➡ Résultat final : 6

👉 Cette méthode permet de :

- comprendre ce que fait réellement le programme ;
- repérer immédiatement les erreurs de raisonnement ;
- expliquer son code à quelqu'un d'autre.

Cas très courant : compter sous condition

Exemple : compter le nombre de lettres "a" dans un mot.

```
mot = "banane"
count = 0

for lettre in mot:
    if lettre == "a":
        count = count + 1

print(count)
```

Ici :

- la boucle parcourt chaque lettre ;
- le compteur n'augmente **que si la condition est vraie**.

À retenir

- Pour **compter**, on ajoute toujours 1 à une variable compteur.
- Pour **cumuler**, on ajoute une valeur qui change à une variable total.
- La variable de comptage ou de cumul :
 - est initialisée **avant** la boucle ;
 - est modifiée **dans** la boucle.
- En cas de doute, **simuler sur papier** avec un tableau est la meilleure stratégie.
- Comprendre l'évolution des variables est plus important que mémoriser le code.

Si tu veux, je peux ensuite :

- proposer **une série d'exercices ciblés "compter / cumuler"**,
- écrire un **article jumeau** sur les erreurs fréquentes,
- ou relier cette notion à des **cas concrets de jeu ou de statistiques**.

Python: Les boucles for sur des listes

Une boucle `for` ne sert pas seulement à répéter N fois la même chose. Sa vraie force, c'est de **parcourir** une collection** d'éléments — une liste de prénoms, de prix, de notes — et de traiter chacun à tour de rôle. C'est ce qu'on apprend ici.

Dans le cours précédent, on a vu `for i in range(N)` pour répéter N fois la même action. Mais quand chaque tour de boucle traite un élément **différent** — un prénom, un prix, un nom d'animal — il faut une **liste**. C'est l'outil de stockage le plus basique de Python, et il marche main dans la main avec `for`.

Objectifs

À la fin de ce cours, tu seras capable de :

1. Créer une **liste** Python avec les crochets `[ ]`.
2. **Parcourir** une liste avec une boucle `for`.
3. Appliquer la **convention de nommage** : nom de la liste au pluriel, variable d'itération au singulier.
4. Faire varier le traitement dans la boucle (afficher, modifier, conditionner).
5. Reconnaître qu'une liste peut contenir n'importe quel type de valeurs.

Partie 1 — Le problème : un truc différent à chaque tour

Reprends l'idée du cours précédent. On voulait afficher des prénoms d'élèves :

```
print("Alice")
print("Bob")
print("Charlie")
print("Diane")
print("Eli")
```

Avec ce qu'on a vu jusqu'ici (`for + range`), on **ne sait pas faire** : `for i in range(5)` répèterait 5 fois la **même** ligne. Mais nous, on veut **5 lignes différentes**.

Il manque un endroit où **stocker les cinq prénoms à l'avance**. C'est exactement le rôle d'une **liste**.

Partie 2 — Créer une liste

Une liste se crée avec des **crochets** `[ ]` et des éléments séparés par des virgules :

```
prenoms = ["Alice", "Bob", "Charlie", "Diane", "Eli"]
```

Quelques détails à remarquer :

- Le nom de la variable est **au pluriel** : `prenoms` . C'est une convention universelle en Python (et dans la plupart des langages), parce que la variable contient **plusieurs valeurs**.
- Chaque élément est entre **guillemets** : on stocke des chaînes de caractères.
- L'ordre est conservé : `prenoms[0]` vaut `"Alice"` , `prenoms[1]` vaut `"Bob"` , etc.

💡 **Vocabulaire** : un élément d'une liste s'appelle aussi un **item** (GB **item**). On dit « parcourir les items » ou « traiter chaque item ».

Partie 3 – Parcourir une liste avec `for`

C'est ici que tout se met en place. Pour traiter chaque prénom de la liste à tour de rôle :

```
prenoms = ["Alice", "Bob", "Charlie", "Diane", "Eli"]

for prenom in prenoms:
    print(prenom)
```

Résultat :

```
Alice
Bob
Charlie
Diane
Eli
```

Exactement ce qu'on voulait — **et la boucle s'adapte automatiquement** au nombre d'éléments. Si tu ajoutes 195 pré-noms à la liste, la boucle s'exécute 200 fois sans rien changer d'autre.

► La convention de nommage : pluriel / singulier

Regarde bien les deux noms de variables dans la boucle :

```
for prenom in prenoms:
#   |         |
#   |         |
# singulier  pluriel
# (un item)  (la collection)
```

C'est **la** convention Python pour les boucles `for` sur listes. Elle se lit en français naturel :

« Pour chaque **prénom** dans la liste des **prénoms**, fais... »

Autres exemples :

LISTE (PLURIEL)	VARIABLE DE BOUCLE (SINGULIER)	LECTURE
<code>animaux</code>	<code>animal</code>	« Pour chaque animal dans animaux »
<code>prix</code>	<code>prix_unitaire</code> (ou <code>p</code>)	« Pour chaque prix dans prix »
<code>notes</code>	<code>note</code>	« Pour chaque note dans notes »
<code>eleves</code>	<code>eleve</code>	« Pour chaque élève dans élèves »
<code>voitures</code>	<code>voiture</code>	« Pour chaque voiture dans voitures »

 **Pourquoi cette convention ?** Un lecteur qui découvre ton code doit pouvoir comprendre en deux secondes la nature de chaque variable. `for animal in animaux` est instantanément clair. `for x in liste1` est obscur. Adopte le pluriel/singulier dès le début, c'est gratuit et ça change la lisibilité.

Partie 4 – Que faire dans la boucle ?

À l'intérieur de la boucle, tu peux faire **tout ce que tu veux** avec l'élément courant. Voici les usages les plus fréquents.

► 4.1 – Afficher chaque élément

C'est l'exemple qu'on vient de voir :

```
animaux = ["Chat", "Lapin", "Capibara", "Dragon de Komodo"]

for animal in animaux:
    print(animal)
```

► 4.2 – Afficher avec une mise en forme

```
animaux = ["Chat", "Lapin", "Capibara"]

# On utilise une f-string pour entourer le nom de texte
for animal in animaux:
    print(f"🐾 {animal} a été ajouté au zoo.")
```

Résultat :

- 🐾 Chat a été ajouté au zoo.
- 🐾 Lapin a été ajouté au zoo.
- 🐾 Capibara a été ajouté au zoo.

► 4.3 – Transformer chaque élément

```
prenoms = ["alice", "bob", "charlie"]  
  
# La méthode .upper() met une chaîne en majuscules  
for prenom in prenoms:  
    print(prenom.upper())
```

Résultat :

```
ALICE  
BOB  
CHARLIE
```

► 4.4 – Conditionner le traitement

Tu peux combiner `for` avec un `if` pour ne traiter qu'une partie des éléments :

```
notes = [12, 8, 17, 4, 15, 19, 11]  
  
# On affiche uniquement les notes en échec (< 10)  
for note in notes:  
    if note < 10:  
        print(f"⚠ Note insuffisante : {note}/20")
```

Résultat :

```
⚠ Note insuffisante : 8/20  
⚠ Note insuffisante : 4/20
```

💡 **Tu remarques ?** La boucle parcourt **toutes** les notes, mais l' `if` filtre celles qu'on veut afficher. C'est la combinaison la plus utilisée en pratique.

Partie 5 – Une liste peut contenir n'importe quoi

Les listes ne sont pas limitées aux chaînes. Tu peux y mettre des nombres, des booléens, des choses mélangées.

► 5.1 – Liste de nombres

```
prix = [2.50, 4.99, 1.20, 8.75, 3.00]

# Affichage formaté
for p in prix:
    print(f"{p:.2f} €")
```

Résultat :

```
2.50 €
4.99 €
1.20 €
8.75 €
3.00 €
```

 **Le `:.2f` dans la f-string** force 2 chiffres après la virgule. Très pratique pour les montants.

► 5.2 – Liste de booléens

```
reponses = [True, False, True, True, False]

for reponse in reponses:
    if reponse:
        print("✓ Correct")
    else:
        print("X Faux")
```

► 5.3 – Liste hétérogène (à éviter en général)

Python autorise des listes avec des types mélangés :

```
truc = ["Alice", 17, True, 3.14]

for element in truc:
    print(element)
```

Ça fonctionne, mais **évite-le** : une liste qui mélange tout, c'est un signe qu'on aurait dû utiliser une autre structure (un dictionnaire, par exemple — on verra plus tard).

 **Bonne pratique** : une liste = une collection d'éléments du même type, qui représentent la même chose.



Exercices

► Exercice guidé — Le panier de courses

Énoncé. Tu as la liste des courses à faire :

```
courses = ["pain", "lait", "œufs", "fromage", "tomates"]
```

Écris un programme qui affiche :

À acheter :

- pain
- lait
- œufs
- fromage
- tomates

Total : 5 articles

ÉTAPE 1 — IDENTIFIER CE QU'IL FAUT

Trois choses à faire :

1. Afficher une ligne d'en-tête (`À acheter :`).
2. Parcourir la liste et afficher chaque article précédé d'un `-`.
3. Afficher une ligne de total.

Le **point 2** est le cœur de la boucle.

ÉTAPE 2 — LA BOUCLE QUI AFFICHE LES ARTICLES

```
courses = ["pain", "lait", "œufs", "fromage", "tomates"]

for course in courses:
    print(f"- {course}")
```

Note le **pluriel/singulier** : `for course in courses` .

Résultat à ce stade :

- pain
- lait
- œufs
- fromage
- tomates

ÉTAPE 3 — AJOUTER L'EN-TÊTE ET LE TOTAL

L'en-tête se met **avant** la boucle, le total **après**. La fonction `len()` donne la taille d'une liste.

```
courses = ["pain", "lait", "œufs", "fromage", "tomates"]

print("À acheter :")

for course in courses:
    print(f"- {course}")

print(f"\nTotal : {len(courses)} articles")
```

Le `\n` au début de la dernière f-string ajoute une ligne vide avant le total.

Résultat final :

```
À acheter :
- pain
- lait
- œufs
- fromage
- tomates

Total : 5 articles
```

 **Le squelette à retenir** : en-tête → boucle → bilan. C'est un pattern qu'on retrouve partout (factures, bulletins, statistiques...).

► Exercice 2 – Le zoo

Tu as la liste suivante :

```
animaux = ["Chat", "Lapin", "Capibara", "Lion", "Dragon de Komodo"]
```

Écris un programme qui affiche pour chaque animal :

```
🐾 Chat fait partie du zoo. (4 lettres)
🐾 Lapin fait partie du zoo. (5 lettres)
🐾 Capibara fait partie du zoo. (8 lettres)
🐾 Lion fait partie du zoo. (4 lettres)
🐾 Dragon de Komodo fait partie du zoo. (16 lettres)
```

Indice : `len(animal)` te donne le nombre de caractères du nom (espaces compris).

► Exercice 3 – Notes au-dessus de la moyenne

Pour la liste suivante :

```
notes = [11, 7, 15, 18, 4, 13, 9, 16]
```

Affiche **uniquement** les notes ≥ 10 , dans ce format :

```
✓ 11/20
✓ 15/20
✓ 18/20
✓ 13/20
✓ 16/20
```

Indice : combine `for` et un `if` .

► Exercice 4 – Inventaire avec prix

À partir des listes suivantes (qu'on suppose alignées dans le même ordre – produit $i \leftrightarrow$ prix i) :

```
produits = ["Pain", "Lait", "Œufs", "Fromage"]
prix      = [1.50, 1.20, 2.80, 4.99]
```

Écris un programme qui affiche :

```
- Pain      : 1.50 €
- Lait      : 1.20 €
- Œufs      : 2.80 €
- Fromage   : 4.99 €
```

Indice : ici on a deux listes à parcourir en parallèle. Utilise `range(len(produits))` et accède aux deux listes par leur index – ou découvre seul la fonction `zip()` si tu es curieux.

À retenir

- Une **liste** se crée avec `[ ]` : `prenoms = ["Alice", "Bob"]` .
- Une boucle `for` parcourt **tous les éléments**, un par un, dans l'ordre.
- **Convention de nommage** : liste au **pluriel** (`animaux`), variable d'itération au **singulier** (`animal`).
- À l'intérieur de la boucle, tu peux **afficher**, **transformer**, **conditionner** – tout ce que tu veux faire avec l'élément courant.
- La boucle s'adapte **automatiquement** à la taille de la liste : pas besoin d'écrire combien d'éléments.
- Une liste peut contenir n'importe quel type, mais **par bonne pratique** elle contient des éléments de même nature.
- `len(liste)` te donne le nombre d'éléments.

Suite

Tu sais maintenant parcourir une liste. Une application directe très fréquente : **afficher un menu de sélection à l'utilisateur** — par exemple les choix d'un jeu, les rayons d'un magasin, les options d'une appli. C'est l'objet du cours suivant, qui montre la progression *menu manuel* → *menu avec une liste* → *menu numéroté avec* `enumerate()` .

Python: Afficher un menu de sélection

*Un menu dans une console, c'est l'application la plus fréquente du combo *liste + boucle for*. On part d'un menu écrit à la main, on le remplace par un menu généré depuis une liste, puis on lui ajoute des numéros automatiques avec `enumerate()`. Et à la fin, l'utilisateur peut vraiment choisir.*

Presque toutes les applications console montrent un menu à un moment ou l'autre : un jeu (« Nouvelle partie / Charger / Quitter »), une appli de quiz (« Choisis ta catégorie »), un script utilitaire (« Que veux-tu faire ? »). Ce cours montre la **bonne façon** de l'écrire — et surtout, comment passer du « ça marche pour 3 options » au « ça marche pour 30 ».

Objectifs

À la fin de ce cours, tu seras capable de :

1. Écrire un menu console avec un titre, des options et un choix utilisateur.
2. Reconnaître pourquoi le menu « écrit à la main » devient ingérable au-delà de 3-4 options.
3. Générer un menu **depuis une liste**, avec une boucle `for`.
4. Ajouter une **numérotation automatique** grâce à `enumerate()`.
5. Récupérer le choix de l'utilisateur et l'utiliser pour accéder à l'option sélectionnée.

Partie 1 — Version 1 : le menu écrit à la main

Imagine que tu codes un mini-zoo et que tu dois afficher la liste des animaux disponibles pour que l'utilisateur en choisisse un. Première approche, naïve, mais qui marche :

```
print("Choisis un animal :")
print("Chat")
print("Lapin")
print("Capibara")
```

Résultat :

```
Choisis un animal :
Chat
Lapin
Capibara
```

Ça marche pour 3 animaux. Mais regarde ce qui cloche déjà :

- **Pas de numéros** : comment l'utilisateur exprime son choix ? Il doit taper le mot exact ? Avec ou sans accent ?
- **Pas évolutif** : si on ajoute un animal, il faut ajouter une ligne `print` .
- **Couplage fort** : le menu est figé dans le code, on ne peut pas le générer depuis un fichier ou une saisie.

💡 **Le symptôme** : plusieurs `print` qui se ressemblent comme deux gouttes d'eau. C'est exactement le signal qui dit « remplace-moi par une boucle ! ».

Partie 2 – Version 2 : le menu via une liste + `for`

Première amélioration : ranger les animaux dans une **liste** et les afficher avec une boucle.

```
# La liste des animaux du zoo
animaux = ["Chat", "Lapin", "Capibara"]

print("Choisis un animal :")

# La boucle affiche un animal par ligne
for animal in animaux:
    print(animal)
```

Résultat identique à la version 1 :

```
Choisis un animal :
Chat
Lapin
Capibara
```

À première vue, c'est plus long que la version 1. **Mais** :

- **Pour ajouter un animal**, il suffit de l'ajouter à la liste : `animaux = ["Chat", "Lapin", "Capibara", "Lion"]` . La boucle l'affichera tout seul.
- **Pour 30 animaux**, ce code fait toujours 5 lignes.
- **Si la liste vient d'ailleurs** (un fichier, une base de données, une saisie utilisateur), seul `animaux = ...` change. Le menu reste le même.

💡 **Convention** rappelée du cours précédent : la liste est au **pluriel** (`animaux`), la variable de boucle au **singulier** (`animal`). Tu peux lire le code à voix haute : « pour chaque animal dans animaux, affiche-le ».

C'est mieux, mais l'utilisateur ne sait toujours pas quoi taper pour choisir. Il faut un **numéro** devant chaque option.

Partie 3 – Version 3 : numéroté avec `enumerate()`

Ce qu'on veut afficher maintenant :

Choisis un animal :

1. Chat
2. Lapin
3. Capibara

Pour avoir un numéro qui s'incrémente automatiquement à chaque tour, Python fournit la fonction `enumerate()`. Elle prend une liste et te donne, à chaque tour, **deux valeurs** : l'index et l'élément.

```
animaux = ["Chat", "Lapin", "Capibara"]

print("Choisis un animal :")

for (i, animal) in enumerate(animaux):
    print(f"{i}. {animal}")
```

Résultat :

Choisis un animal :

0. Chat
1. Lapin
2. Capibara

Tu remarques quelque chose : la numérotation commence à **0**. C'est parce que Python compte à partir de 0 (l'index d'un élément dans une liste). Pour un humain, on préfère commencer à 1.

► Décortiquons `enumerate()`

```
for (i, animal) in enumerate(animaux):
#   |         |         |
#   |         |         |
# deux variables   enumerate produit
# par tour         des paires (index, élément)
```

À chaque tour de boucle, `enumerate` donne une **paire** :

TOUR	I	ANIMAL
1	0	"Chat"
2	1	"Lapin"
3	2	"Capibara"

i Sans `enumerate()` tu aurais dû gérer un compteur à la main : `i = 0; for animal in animaux: print(i, animal); i += 1`. C'est plus long, plus fragile, et personne ne fait ça en Python — `enumerate` est l'outil dédié.

► Commencer à 1 au lieu de 0

Deux possibilités :

SOLUTION A — AJOUTER 1 DANS LE `PRINT`

```
for (i, animal) in enumerate(animaux):  
    print(f"{i + 1}. {animal}")
```

SOLUTION B — LE PARAMÈTRE `START` DE `ENUMERATE()`

```
for (i, animal) in enumerate(animaux, start=1):  
    print(f"{i}. {animal}")
```

Les deux donnent le même résultat. La solution B est légèrement plus propre — `start=1` exprime clairement l'intention.

Choisis un animal :

1. Chat
2. Lapin
3. Capibara

Partie 4 — Récupérer le choix de l'utilisateur

Le menu est joli, mais à quoi sert-il si l'utilisateur ne peut pas répondre ? Ajoutons la **saisie**.

```
animaux = ["Chat", "Lapin", "Capibara"]  
  
print("Choisis un animal :")  
  
for (i, animal) in enumerate(animaux, start=1):  
    print(f"{i}. {animal}")  
  
# Récupère la réponse, convertit en entier  
choix = int(input("Ton choix ? "))  
  
# Retrouve l'animal correspondant (attention : -1 car les listes commencent à 0)  
animal_choisi = animaux[choix - 1]  
  
print(f"Tu as choisi : {animal_choisi}")
```

Déroulement à l'exécution :

```
Choisis un animal :  
1. Chat  
2. Lapin  
3. Capibara  
Ton choix ? 2  
Tu as choisi : Lapin
```

⚠ **Pourquoi** `choix - 1` ? L'utilisateur tape `2` pour le deuxième animal. Mais dans la liste, le deuxième animal est à l'index `1` (les listes commencent à 0). D'où le `-1`. Ne te trompe pas dans ce petit décalage, c'est la source d'erreur n°1 sur les menus.

► Cas problématique : que se passe-t-il si l'utilisateur tape n'importe quoi ?

- Il tape `0` → `animaux[-1]` → renvoie le **dernier** animal (`Capibara`). Comportement bizarre mais pas plantage.
- Il tape `99` → `animaux[98]` → **plantage** (`IndexError`).
- Il tape `chien` → `int("chien")` → **plantage** (`ValueError`).

Pour gérer ces cas proprement, il faut ajouter une **validation** — c'est typiquement le rôle d'une boucle `while` qui demande tant que la réponse est invalide. On voit ça dans le cours sur les boucles `while` avec validation.

Partie 5 — Le programme complet, commenté

Voici la version finale, avec commentaires pédagogiques :

```
# --- Données du menu (modifie cette liste pour changer le menu) ---
animaux = ["Chat", "Lapin", "Capibara", "Lion", "Dragon de Komodo"]

# --- Affichage du menu ---
print("=== ZOO DU LYCÉE ===")
print("Choisis un animal à observer :")
print() # ligne vide

for (i, animal) in enumerate(animaux, start=1):
    print(f" {i}. {animal}")

print()

# --- Saisie du choix ---
choix = int(input("Ton choix ? "))

# --- Traitement du choix ---
animal_choisi = animaux[choix - 1]
print(f"\n→ Tu pars observer un {animal_choisi}. Bonne visite !")
```

Exécution :

```
=== ZOO DU LYCÉE ===
Choisis un animal à observer :

1. Chat
2. Lapin
3. Capibara
4. Lion
5. Dragon de Komodo

Ton choix ? 3

→ Tu pars observer un Capibara. Bonne visite !
```

Note que pour **ajouter un animal**, il suffit de l'ajouter à la liste `animaux`. Tout le reste du programme s'adapte tout seul. **C'est ça, l'élégance d'un menu géré par boucle.**



Exercices

► Exercice guidé — Le menu du restaurant

Énoncé. Écris un programme qui :

1. Affiche le menu d'un restaurant avec ces plats : ["Lasagnes", "Salade César", "Burger", "Pizza margherita", "Tarte aux fruits"]
2. Numérote chaque plat à partir de 1.
3. Demande à l'utilisateur son choix.
4. Affiche un message du type : → Commande validée : Pizza margherita.

ÉTAPE 1 – RÉFLÉCHIR AU PLAN

Ce qu'il faut :

- Une **liste** des plats.
- Une **boucle for + enumerate** pour afficher avec numéros.
- Un **input** pour le choix.
- Un accès à la liste avec `[choix - 1]` pour récupérer le plat.

ÉTAPE 2 – LE SQUELETTE

```
plats = ["Lasagnes", "Salade César", "Burger", "Pizza margherita", "Tarte aux fruits"]

print("Menu du restaurant :")
# TODO : boucle d'affichage

# TODO : saisie du choix

# TODO : affichage de la commande
```

ÉTAPE 3 – LA BOUCLE D'AFFICHAGE

```
for (i, plat) in enumerate(plats, start=1):
    print(f" {i}. {plat}")
```

(Convention pluriel/singulier : `plats` → `plat` .)

ÉTAPE 4 – LA SAISIE

```
choix = int(input("\nQuel plat veux-tu ? "))
```

ÉTAPE 5 – LA VALIDATION FINALE

```
plat_choisi = plats[choix - 1]
print(f"→ Commande validée : {plat_choisi}.")
```

ÉTAPE 6 – LE PROGRAMME COMPLET

```
plats = ["Lasagnes", "Salade César", "Burger", "Pizza margherita", "Tarte aux fruits"]

print("Menu du restaurant :")

for (i, plat) in enumerate(plats, start=1):
    print(f" {i}. {plat}")

choix = int(input("\nQuel plat veux-tu ? "))
plat_choisi = plats[choix - 1]

print(f"→ Commande validée : {plat_choisi}.")
```

Exécution :

```
Menu du restaurant :
 1. Lasagnes
 2. Salade César
 3. Burger
 4. Pizza margherita
 5. Tarte aux fruits

Quel plat veux-tu ? 4
→ Commande validée : Pizza margherita.
```

► Exercice 2 – Le menu d'un jeu

Écris un menu de jeu vidéo avec les options :

```
options = ["Nouvelle partie", "Charger une partie", "Options", "Crédits", "Quitter"]
```

Le menu doit ressembler à ça :

```
┌──────────────────┐
│ MENU PRINCIPAL   │
└──────────────────┘

1. Nouvelle partie
2. Charger une partie
3. Options
4. Crédits
5. Quitter

Ton choix > 1
Tu as choisi : Nouvelle partie
```

► Exercice 3 — Menu de classes du lycée

À partir de la liste :

```
classes = ["3TTR", "4TTR", "5TTR", "6TTR", "5TQ", "6TQ", "3GMS", "4GMS", "5GMS", "6GMS"]
```

Écris un programme qui demande à l'élève sa classe et affiche le message :

```
Classes disponibles :
```

```
1. 3TTR
```

```
2. 4TTR
```

```
...
```

```
10. 6GMS
```

```
Ta classe ? 5
```

```
→ Bienvenue en 5TTR !
```

► Exercice 4 — Bonus : éviter la duplication

Tu veux pouvoir afficher **plusieurs menus différents** dans ton programme (menu principal, sous-menu options, etc.) — sans copier-coller la logique d'affichage. Écris une fonction `afficher_menu(titre, options)` qui prend un titre et une liste, et fait l'affichage. Indice : c'est juste la mise en commun de tout ce qu'on a vu, dans un `def`. (Si tu n'as pas encore vu les fonctions, garde cet exercice pour plus tard.)



À retenir

- Un menu console se construit avec **trois briques** : une liste d'options, une boucle `for + enumerate`, un `input`.
- `enumerate(liste)` donne à chaque tour une paire `(index, élément)` — utile dès qu'il faut un numéro devant les éléments.
- `enumerate(liste, start=1)` commence la numérotation à 1 — plus naturel pour un humain.
- Pour récupérer l'option choisie : `options[choix - 1]` (attention au `-1` puisque les listes commencent à 0).
- Pour ajouter ou retirer une option, on **modifie la liste** — rien d'autre ne change.
- La gestion des erreurs de saisie (lettres au lieu de numéros, numéro hors limites) demande une boucle `while` avec validation — sujet du cours suivant.

Suite

Le menu fonctionne, mais il **plante** si l'utilisateur tape n'importe quoi. Pour gérer ça proprement — redemander tant que la saisie est invalide — il nous faut une autre famille de boucles : **les boucles** `while`. On y vient.

Python: Les fonctions

En Python, comme dans d'autres langages de programmation, une fonction est une suite d'instructions que l'on peut appeler avec un nom. C'est un moyen de regrouper des opérations pour les réutiliser efficacement. Cela permet de rendre ton code plus clair, plus organisé et plus facile à maintenir.

Qu'est-ce qu'une fonction?

Une fonction est une **séquence d'instructions** que Python effectue lorsqu'elle est appelée. Elle peut prendre des **paramètres** et peut **renvoyer une valeur**. Les fonctions sont définies avec le mot-clé `def` suivi du nom de la fonction et de parenthèses qui *peuvent* contenir des paramètres.

► Syntaxe de base

```
def nom_de_la_fonction(paramètre1, paramètre2, ...):  
    # Bloc d'instructions  
    return valeur # Facultatif
```

Pourquoi utiliser des fonctions?

- **Réutilisation de code:** Écris une fois, utilise plusieurs fois.
- **Organisation:** Divise ton programme en petites parties gérables.
- **Maintenance:** Corrige ou améliore une partie sans toucher au reste.
- **Clarté:** Rend ton code plus lisible et compréhensible.

Créer une fonction simple

► Exemple:

```
def saluer(nom):  
    print(f"Bonjour, {nom}!")
```

► Appel de la fonction:

```
saluer("Alice")
```

► Résultat:

```
Bonjour, Alice!
```

Paramètres et arguments

- **Paramètres:** Variables listées dans la définition de la fonction.
- **Arguments:** Valeurs fournies lors de l'appel de la fonction.

► Types d'arguments

- **Positionnels:** Doivent être passés dans l'ordre.
- **Nominaux (mot-clé):** Peuvent être passés dans n'importe quel ordre en spécifiant leur nom.

► Exemple avec arguments nominaux:

```
def afficher_info(nom, age):  
    print(f"Nom: {nom}, Âge: {age}")  
  
afficher_info(age=30, nom="Alice")
```

Valeurs de retour

Utilise `return` pour renvoyer une valeur à l'endroit où la fonction est appelée.

► Exemple:

```
def somme(a, b):  
    return a + b  
  
resultat = somme(5, 3)  
print(resultat) # Affichera 8
```

Tu peux faire ce que tu veux avec la valeur d'une fonction: l'afficher, l'utiliser dans un calcul, la stocker dans une variable (comme dans l'exemple précédent)... ou encore... ne rien faire:

```
print(resultat(2,5)) # Affichera 7  
print(2 * resultat(5,5)) # Affichera 20  
resultat(7,7) # Pas très utile, mais fonctionne
```

Portée des variables

Les variables en Python peuvent être soit locales, soit globales, et la différence entre les deux est cruciale pour comprendre comment Python gère l'espace de noms des variables

- **Locale:** Existe uniquement à l'intérieur de la fonction.
- **Globale:** Accessible partout dans le code.

Une variable **locale** est définie **à l'intérieur** d'une fonction et n'existe que dans l'espace local de cette fonction. Elle est **inaccessible depuis l'extérieur de la fonction**, ce qui signifie que toute tentative d'y accéder en dehors de la fonction entraînera une erreur. Cela permet de s'assurer que la fonction ne dépend pas de, ou n'interfère pas avec, le reste du programme.

À l'inverse, une variable **globale** est définie en dehors de toutes les fonctions et est **accessible depuis n'importe quelle partie du programme**, y compris à l'intérieur des fonctions. Cependant, si tu souhaites modifier une variable globale à l'intérieur d'une fonction, tu dois d'abord la déclarer comme `global` dans la fonction; sinon, Python créera une nouvelle variable locale du même nom. Il est généralement conseillé d'utiliser des variables locales autant que possible pour éviter des effets secondaires inattendus et rendre le code plus facile à comprendre et à maintenir.

► Exemple de variable locale:

```
def exemple():  
    x = 5  # x est local à exemple  
  
exemple()  
print(x)
```

Bonnes pratiques

- **Nom explicite:** Choisir des noms de fonctions clairs et descriptifs.
- **Une tâche:** Chaque fonction doit avoir une seule responsabilité.
- **Courte:** Si une fonction devient trop longue, envisage de la diviser.
- **Documentation:** Utilise des docstrings pour expliquer ce que fait la fonction.

► Exemple de docstring:

```
def somme(a, b):  
    """  
    Retourne la somme de deux nombres.  
    """  
    return a + b
```

Conclusion

Les fonctions sont un outil puissant en Python qui t'aide à écrire des codes clairs, organisés et réutilisables. En les utilisant judicieusement, tu peux rendre la programmation plus agréable et maintenir tes projets plus facilement. Pense à les utiliser chaque fois que tu remarques une répétition ou que tu souhaites clarifier ton code!

Python: Les fonctions (pratique autonome)

En Python, comme dans d'autres langages de programmation, une fonction est une suite d'instructions que l'on peut appeler avec un nom. C'est un moyen de regrouper des opérations pour les réutiliser efficacement. Cela permet de rendre ton code plus clair, plus organisé et plus facile à maintenir.

Envoyez ces exercices dans le dossier `Python / Exercices / Fonctions / 0 - Pratique autonome`.

► Exercice 11: Conversion de miles en kilomètres

Objectif: Pratiquer la conversion simple.

Énoncé: Crée une fonction `miles_en_kilometres(miles)` qui convertit les miles en kilomètres en utilisant la formule 1 mile = 1.60934 kilomètres.

Exemple:

```
print(miles_en_kilometres(5))
```

Sortie attendue:

```
8.0467
```

Nom du fichier: `fonctions-11-miles-[prenom].py`

► Exercice 12: Température de surface d'une étoile

Objectif: Utiliser la loi de Wien pour des calculs astronomiques.

Énoncé: La loi de Wien relie la température d'une étoile à la longueur d'onde à laquelle elle brille le plus. Écris une fonction `temperature_etoile(longueur_max)` qui calcule la température de surface approximative de l'étoile en Kelvin. Utilise la formule $T = \frac{2.898 \cdot 10^6}{longueur_{max}}$ où `longueur_max` est en micromètres

Exemple:

```
print(temperature_etoile(0.5))
```

Sortie attendue:

```
5796.0 K
```

Nom du fichier: `fonctions-12-etoile-[prenom].py`

► Exercice 13: Conversion de radians en degrés

Objectif: Travailler avec des conversions d'unités angulaires.

Énoncé: Définis une fonction `radians_en_degrés(radians)` qui convertit une mesure en radians en degrés en utilisant la formule $\text{degrés} = \text{radians} * \frac{180}{\pi}$.

Exemple:

```
import math
print(radians_en_degrés(math.pi/4))
```

Sortie attendue:

```
45.0
```

Nom du fichier: `fonctions-13-radians-[prenom].py`

► Exercice 14: Calcul de la force gravitationnelle

Objectif: Appliquer la loi de la gravitation universelle de Newton.

Énoncé: Écris une fonction `force_gravitationnelle(m1, m2, r)` qui calcule la force gravitationnelle entre deux masses (en kg) à une certaine distance (en mètres). Utilise la formule $F = G \frac{m1 * m2}{r^2}$, où **G** est la constante gravitationnelle $6.674 * 10^{-11} \text{ m}^3\text{kg}^{-1}\text{s}^{-2}$.

Exemple:

```
print(force_gravitationnelle(5.98e24, 1.99e30, 1.50e11)) # Terre - Soleil
```

Sortie attendue:

```
3.5277481777777787e+22
```

Nom du fichier: `fonctions-14-radians-[prenom].py`

► Exercice 15: Conversion de pression en Pascal

Objectif: Pratiquer les conversions d'unités de pression.

Énoncé: Crée une fonction `atm_en_pascal(atm)` qui convertit la pression atmosphérique de atmosphères en Pascal. Utilise la formule $1\text{atm} = 101325 \text{ Pa}$.

Exemple:

```
print(atm_en_pascal(1))
```

Sortie attendue:

Nom du fichier: `fonctions-15-pression-[prenom].py`

► Exercice 16: Calcul du volume d'un cylindre

Objectif: Appliquer des formules géométriques simples.

Énoncé: Écris une fonction `volume_cylindre(rayon, hauteur)` qui calcule le volume d'un cylindre en utilisant la formule $V = \pi * rayon^2 * hauteur$.

Exemple:

```
import math
print(volume_cylindre(3, 5))
```

Sortie attendue:

```
Valeur du volume en unités cubiques
```

Nom du fichier: `fonctions-16-cylindre-[prenom].py`

► Exercice 17: Conversion d'énergie cinétique

Objectif: Comprendre la relation entre différentes unités d'énergie.

Énoncé: Définis une fonction `joules_en_kilowattheures(joules)` qui convertit l'énergie en joules en kilowatt-heures, sachant que $1 \text{ kWh} = 3.6 * 10^6 \text{ joules}$.

Exemple:

```
print(joules_en_kilowattheures(3600000))
```

Sortie attendue:

```
1.0
```

Nom du fichier: `fonctions-17-joules-[prenom].py`

► Exercice 18: Conversion de décibels en intensité sonore

Objectif: Travailler avec des logarithmes dans des conversions.

Énoncé: Crée une fonction `decibels_en_intensite(decibels)` qui convertit des niveaux de décibels en intensité sonore en Watts par mètre carré en utilisant la formule $I = 10^{\frac{\text{decibels}}{10}} * 10^{-12}$.

Exemple:

```
print(decibels_en_intensite(90))
```

Sortie attendue:

Valeur de l'intensité en W/m²

Nom du fichier: `fonctions-18-decibels-[prenom].py`

► Exercice 19: Distance de freinage d'un robot

Objectif: Appliquer des concepts de physique de base à la robotique.

Énoncé: Écris une fonction `distance_freinage(vitesse, coefficient)` qui calcule la distance minimale nécessaire pour qu'un robot s'arrête complètement. Utilise la formule $distance = \frac{vitesse^2}{2 * coefficient * g}$, où *vitesse* est en m/s, *coefficient* est le coefficient de friction entre les roues et le sol, et *g* est l'accélération due à la gravité 9.81m/s².

Exemple:

```
print(distance_freinage(10, 0.7))
```

Sortie attendue:

Valeur de la distance en mètres

Nom du fichier: `fonctions-19-freinage-[prenom].py`

Python: Les fonctions (pratique guidée)

En Python, comme dans d'autres langages de programmation, une fonction est une suite d'instructions que l'on peut appeler avec un nom. C'est un moyen de regrouper des opérations pour les réutiliser efficacement. Cela permet de rendre ton code plus clair, plus organisé et plus facile à maintenir.

Envoyez ces exercices dans le dossier `Python / Exercices / Fonctions / 01 - Pratique guidée` .

Exercice 1: Fonction de Bienvenue

Objectif: Créer une fonction simple.

Énoncé: Écris une fonction `bienvenue()` qui prend un nom en argument et affiche "Bienvenue, [Nom] !" à l'écran.

Exemple:

```
bienvenue("Alice")
```

Sortie attendue:

```
Bienvenue, Alice !
```

Nom du fichier: `fonctions-01-bienvenue-[prenom].py`

Exercice 2: Calcul de l'aire d'un rectangle

Objectif: Utiliser des paramètres et retourner une valeur.

Énoncé: Définis une fonction `aire_rectangle(longueur, largeur)` qui calcule et retourne l'aire d'un rectangle.

Exemple:

```
print(aire_rectangle(5, 3))
```

Sortie attendue:

```
15
```

Nom du fichier: `fonctions-02-rectangle-[prenom].py`

Exercice 3: Conversion de température

Objectif: Travailler avec des opérations mathématiques dans une fonction.

Énoncé: Écris une fonction `fahrenheit(celsius)` qui convertit une température donnée de Celsius en Fahrenheit et retourne le résultat. La formule est $(\text{celsius} * 9/5) + 32$. Écris une autre fonction `celsius(fahrenheit)` qui convertit une température donnée de Fahrenheit en Celsius et retourne le résultat. La formule est $(\text{fahrenheit} - 32) * 5/9$.

Exemple:

```
print("F°:", fahrenheit(0))
print("C°:", celsius(0))
```

Sortie attendue:

```
F°: 32.0
C°: -17.78
```

Nom du fichier: `fonctions-03-temperature-[prenom].py`

Exercice 4: Fonction de vérification

Objectif: Utiliser des structures conditionnelles dans une fonction.

Énoncé: Crée une fonction `est_majeur(age)` qui prend un âge en argument et retourne `True` si l'âge est supérieur ou égal à 18, et `False` sinon.

Exemple:

```
print(est_majeur(21))
print(est_majeur(16))
```

Sortie attendue:

```
True
False
```

Nom du fichier: `fonctions-04-majeur-[prenom].py`

Exercice 5: Manipulation de listes

Objectif: Travailler avec des listes et des boucles.

Énoncé: Définis une fonction `moyenne(liste)` qui prend une liste de nombres et retourne la moyenne de ces nombres.

Exemple:

```
notes = [12, 15, 11, 13, 14, 18]
print(moyenne(notes))
```

Sortie attendue:

```
La moyenne est : 13.833333333333334
```

Nom du fichier: `fonctions-05-moyenne-[prenom].py`

Ces exercices couvrent un éventail de compétences de base en Python, de la création de fonctions simples à l'utilisation de structures de contrôle et de types de données plus complexes. Ils sont conçus pour renforcer la compréhension et la pratique de l'écriture et de l'utilisation des fonctions en Python. Bonne chance!

Python: Les modules

Les modules en Python sont des fichiers contenant des définitions et des instructions. Ils permettent de diviser un programme en plusieurs fichiers pour une meilleure organisation et réutilisation du code. Un module peut définir des fonctions, des classes et des variables, ainsi que des blocs de code exécutables.

Qu'est-ce qu'un module?

► Définition

Un module est un fichier contenant du code Python avec l'extension `.py`. Il peut contenir des fonctions, des classes, des variables, et même d'autres modules.

► Avantages

- **Réutilisation de code:** Facilite la réutilisation de code entre différents projets.
- **Organisation:** Permet une meilleure organisation du code en séparant les différentes parties logiques.
- **Espace de noms:** Évite les conflits entre noms de variables, fonctions, etc.

Utilisation des Modules Standards

Python est fourni avec une bibliothèque standard de modules. Voici comment les utiliser :

► Importation

Pour utiliser un module, il faut d'abord l'importer avec le mot-clé `import`.

EXEMPLE:

```
import math
```

► Accès aux Fonctions

Après avoir importé un module, tu peux accéder à ses fonctions et variables en utilisant la notation pointée.

EXEMPLE:

```
print(math.sqrt(16)) # Affiche 4.0
```

Création de tes Propres Modules

► Créer un Fichier Module

1. **Écris du code:** Ouvre un éditeur de texte, écris ton code (par exemple, des fonctions ou des variables).
2. **Enregistre:** Sauvegarde le fichier avec un nom descriptif et l'extension `.py`.

EXEMPLE DE FICHIER `SALUTATIONS.PY` :

```
def bonjour(nom):  
    return f"Bonjour, {nom}!"  
  
def au_revoir(nom):  
    return f"Au revoir, {nom}!"
```

► Utilisation de ton Module

Après avoir créé ton module, tu peux l'importer et utiliser ses fonctions dans d'autres scripts Python.

EXEMPLE:

```
import salutations  
  
print(salutations.bonjour("Alice")) # Affiche "Bonjour, Alice!"
```

Importation Avancée

► Importation Sélective

Si tu veux seulement importer certaines fonctions d'un module, tu peux le faire directement.

EXEMPLE:

```
from math import sqrt  
print(sqrt(16)) # Affiche 4.0
```

► Alias de Module

Tu peux aussi donner un alias aux modules importés pour raccourcir les noms.

EXEMPLE:

```
import numpy as np
```

Chemins de Module

Python recherche les modules dans les répertoires listés dans `sys.path`. Pour ajouter des répertoires à cette liste, tu peux modifier la variable d'environnement `PYTHONPATH` ou utiliser `sys.path.append(chemin)` dans ton script.

► Exemple:

```
import sys
sys.path.append('/chemin/vers/ton/module')
```

Bonnes Pratiques

- **Noms descriptifs:** Donne des noms clairs et descriptifs à tes modules et fonctions.
- **Documentation:** Documente tes modules et fonctions avec des docstrings.
- **Teste ton code:** Écris des tests pour vérifier que ton module fonctionne comme prévu.
- **Respecte PEP 8:** Suis les conventions de style de code de Python pour la lisibilité.

Conclusion

Les modules sont un outil puissant en Python qui te permettent de structurer ton code de manière logique, de réutiliser du code et de partager des fonctionnalités entre différents programmes. En apprenant à les utiliser et à les créer, tu pourras rendre tes projets Python plus organisés, efficaces et faciles à maintenir.

Exercice

Créez un module appelé `utils` avec les fonctions suivantes:

`celsius(temp)` : a function to convert temperatures from fahrenheit to Celsius. `pairs_impairs(liste)` : une fonction qui compte le nombre de nombres pairs et impaires dans une liste et affiche le résultat sur la console. `suite(debut, fin)` : une fonction qui renvoie (`return`) une liste de nombres compris entre `debut` et `fin` (inclus) qui sont divisibles par 7 et par 5.

Créez un fichier Python qui importe votre module et appelle/teste vos fonctions avec 3 cas différents à chaque fois.

Introduction à Python

Python est un langage de programmation interprété, polyvalent et convivial qui a gagné en popularité au fil des ans. Créé par Guido van Rossum et publié pour la première fois en 1991, Python est devenu l'un des langages les plus utilisés dans le monde de la programmation en raison de sa simplicité, de sa lisibilité et de sa grande communauté de développeurs.

Les avantages de Python

► Simplicité et lisibilité

L'une des principales caractéristiques de Python est sa syntaxe simple et concise, qui le rend facile à apprendre et à utiliser même pour les débutants. La lisibilité du code Python est également un avantage majeur, permettant aux développeurs de comprendre rapidement le fonctionnement d'un programme.

► Polyvalence

Python est un langage polyvalent qui peut être utilisé dans une grande variété de domaines, notamment le développement web, l'analyse de données, l'intelligence artificielle, l'automatisation, la robotique et bien plus encore. Sa polyvalence en fait un outil indispensable pour de nombreux développeurs et entreprises.

► Grande bibliothèque standard

Python dispose d'une vaste bibliothèque standard qui offre des fonctionnalités prêtes à l'emploi pour diverses tâches de programmation. Cela permet aux développeurs d'économiser du temps en utilisant des modules et des packages déjà développés plutôt que de réinventer la roue.

► Communauté active

La communauté Python est l'une des plus grandes et des plus actives dans le monde de la programmation. Cette communauté dynamique contribue au développement continu de Python en créant de nouveaux packages, en fournissant des ressources d'apprentissage et en soutenant les nouveaux arrivants.

Qui utilise Python et pourquoi ?

► Développeurs web

Python est largement utilisé dans le développement web pour la création de sites web, d'applications web et de services web. Des frameworks populaires comme Django et Flask permettent de développer rapidement des applications web robustes et évolutives.

► Scientifiques des données

Python est devenu le langage de choix pour de nombreux scientifiques des données en raison de ses puissantes bibliothèques d'analyse de données telles que NumPy, Pandas et Matplotlib. Ces outils facilitent l'exploration, la manipulation et la visualisation des données.

► Ingénieurs en intelligence artificielle et en machine learning

Python est largement utilisé dans le domaine de l'intelligence artificielle et de l'apprentissage automatique en raison de sa facilité d'utilisation et de ses bibliothèques spécialisées telles que TensorFlow, Keras et scikit-learn. Ces bibliothèques permettent de développer et de déployer des modèles d'apprentissage automatique de manière efficace.

► Développeurs de logiciels et d'applications

De nombreuses entreprises utilisent Python pour le développement de logiciels et d'applications en raison de sa polyvalence, de sa productivité et de sa capacité à s'intégrer facilement avec d'autres langages et technologies.

La popularité croissante de Python

Python connaît une popularité croissante grâce à ses nombreux avantages et à sa polyvalence. Selon divers indices de popularité, tels que l'indice TIOBE et l'enquête Stack Overflow Developer Survey, Python figure parmi les langages de programmation les plus populaires et les plus demandés dans l'industrie.

Sa popularité continue de croître en raison de son adoption par de grandes entreprises comme Google, Facebook, Amazon et Netflix, qui utilisent Python pour diverses applications et services.

En conclusion, Python est bien plus qu'un simple langage de programmation. C'est un outil puissant et polyvalent qui continue de gagner en popularité grâce à sa simplicité, sa lisibilité, sa grande communauté et sa polyvalence. Que vous soyez un débutant en programmation ou un développeur expérimenté, Python offre une multitude d'opportunités pour créer des applications innovantes et résoudre des problèmes complexes dans divers domaines.

Notes de cours

- [Télécharger les notes de cours "officielles"](#)

EXERCICES DU JOUR

- [Exercices sur les variables](#)
- [Pour les courageux](#)
- [Exercices sur les conditions](#)
- [Exercices sur les boucles](#)

ELECTRONIQUE

Les actionneurs

Un actionneur est un composant ou un dispositif qui convertit un signal ou une commande électrique, pneumatique ou hydraulique en une action physique ou mécanique. Les actionneurs jouent un rôle essentiel en robotique, car ils permettent aux robots d'agir sur leur environnement et d'exécuter des tâches spécifiques.

Utilité des actionneurs

Les actionneurs sont utilisés pour **générer des mouvements**, des forces ou des **actions physiques** en réponse à des signaux de commande. Ils permettent aux robots **d'interagir activement avec leur environnement** en effectuant des tâches précises. Les actionneurs sont essentiels pour la manipulation d'objets, le déplacement, la propulsion, le mouvement des articulations, le contrôle de la précision, etc.

Les actionneurs permettent aux robots d'**exercer des forces**, de produire des mouvements linéaires ou rotatifs, d'effectuer des changements d'état ou de position, d'appliquer des pressions, de saisir des objets, d'émettre des signaux et bien plus encore. Ils sont les "**moteurs**" qui permettent aux robots de **réaliser des actions concrètes dans le monde réel**.

Différents types d'actionneurs

Il existe plusieurs types d'actionneurs utilisés en robotique. Voici quelques-uns des plus courants :

- **Actionneurs électriques** : Les actionneurs électriques, tels que les **moteurs électriques**, sont alimentés par de l'énergie électrique. Ils peuvent générer des mouvements rotatifs, linéaires ou vibratoires, et sont couramment utilisés dans **les bras robotiques**, les **roues motrices** et d'autres systèmes mécaniques.
- **Actionneurs pneumatiques** : Les actionneurs pneumatiques **utilisent de l'air comprimé** pour générer des forces et des mouvements. Les **vérins pneumatiques** sont un exemple d'actionneur pneumatique couramment utilisé pour produire des mouvements linéaires.
- **Actionneurs hydrauliques** : Les actionneurs hydrauliques **utilisent un fluide** hydraulique pour générer des forces et des mouvements. Les **vérins hydrauliques** sont fréquemment utilisés pour produire des **mouvements linéaires puissants** dans les robots industriels et d'autres applications nécessitant une force élevée.
- **Actionneurs piezoélectriques** : Les actionneurs piezoélectriques utilisent des matériaux piézoélectriques pour convertir une tension électrique en un **mouvement mécanique précis**. Ils sont utilisés dans des applications nécessitant **un contrôle très précis**, tels que les systèmes de positionnement fin et les microrobots.
- **Actionneurs électromagnétiques** : Les actionneurs électromagnétiques **utilisent des champs magnétiques** pour générer des mouvements ou des forces. Les **électroaimants** et les actionneurs à solénoïde sont des exemples courants d'actionneurs électromagnétiques.

Principaux actionneurs utilisés en robotique

- **Moteurs électriques** : Les moteurs électriques, tels que les moteurs à courant continu (DC) et les moteurs pas à pas, sont les actionneurs les plus couramment utilisés en robotique. Ils convertissent l'énergie électrique en mouvement mécanique pour produire des rotations, des translations ou des mouvements précis.
- **Relais**: Les relais sont des composants utilisés pour contrôler des circuits électriques puissants avec de faibles signaux électriques. Ils **fonctionnent comme des interrupteurs**. Ils fonctionnent en utilisant une bobine électromagnétique qui active des contacts électriques pour **fermer ou ouvrir le circuit**. Les relais sont utilisés pour isoler les circuits de commande des circuits de puissance, ce qui permet de contrôler des charges élevées sans endommager les circuits de commande.
- **Vérins pneumatiques** : Les vérins pneumatiques utilisent l'air comprimé pour générer un mouvement linéaire. Ils sont souvent utilisés pour effectuer des mouvements rapides et puissants dans les applications robotiques nécessitant une force importante.
- **Vérins hydrauliques** : Les vérins hydrauliques utilisent un fluide hydraulique sous pression pour générer un mouvement linéaire puissant. Ils sont utilisés dans les applications nécessitant une force élevée et un contrôle précis.
- **Actionneurs électromagnétiques** : Les actionneurs électromagnétiques, tels que les électroaimants et les actionneurs à solénoïde, utilisent des champs magnétiques pour produire un mouvement linéaire ou rotatif. Ils sont couramment utilisés dans les mécanismes de commutation, les systèmes de verrouillage et d'autres applications.

Les 2 types de moteurs principaux

► Moteurs à courant continu (DC)



Les moteurs à courant continu sont largement utilisés en robotique en raison de leur **simplicité**, de leur **coût abordable** et de leur contrôle précis de la **vitesse et de la direction**. Ces moteurs fonctionnent en utilisant un courant continu pour créer un champ magnétique qui génère un **couple de rotation**. Les moteurs DC sont disponibles dans différentes tailles et puissances, ce qui les rend polyvalents pour diverses applications robotiques, allant des petits robots autonomes aux bras robotiques industriels.

C'est ce genre de moteur que l'on utilise pour **faire tourner les roues du robot**.

► Moteurs pas à pas et servomoteurs



Ces moteurs sont utilisés lorsque **la précision de positionnement est primordiale**. Ces moteurs se déplacent par incréments discrets, appelés "pas", en réponse à des signaux de commande spécifiques. Ils sont composés de bobines et d'un rotor magnétique, et fonctionnent en appliquant des impulsions électriques séquentielles aux bobines pour créer un champ magnétique rotatif.

Les moteurs pas à pas offrent un contrôle précis de la position et **peuvent maintenir une position sans consommer d'énergie** supplémentaire. Ils sont couramment utilisés dans les imprimantes 3D, les systèmes de positionnement, les robots de précision et les mécanismes nécessitant un contrôle précis de la rotation.

Il existe des différences entre les moteurs pas à pas et les servomoteurs. la principale différence réside dans la façon dont ces moteurs sont contrôlés et dans leur capacité à maintenir une position précise. **Les servomoteurs sont contrôlés en fonction de la position souhaitée**, tandis que **les moteurs pas à pas se déplacent par incréments précis** en réponse à des signaux de commande spécifiques. Les servomoteurs sont utilisés pour un positionnement précis à **un angle spécifique**, tandis que les moteurs pas à pas offrent une précision de positionnement élevée pour des **mouvements précis** et contrôlés.

► Comparaison

Chaque type de moteur présente des avantages et des limites, et leur sélection dépend des exigences spécifiques de l'application robotique. Les **moteurs à courant continu** sont adaptés aux applications nécessitant un **contrôle de vitesse et de direction**, tandis que les **moteurs pas à pas** sont privilégiés pour leur **précision de positionnement**. Le choix du moteur dépendra donc des besoins spécifiques de conception, de contrôle et de précision du robot.

Conclusion

Chaque type d'actionneur a ses propres caractéristiques, avantages et limites, et est utilisé en fonction des besoins spécifiques du robot et de l'application. L'utilisation appropriée des actionneurs permet aux robots d'interagir avec le monde réel, d'exécuter des tâches et de réaliser des mouvements précis et contrôlés.

Feux de circulation

Cet article explique comment brancher et contrôler les 3 led du module 'feu de signalisation'



Pour contrôler un feu de signalisation (traffic lights) à l'aide d'une Raspberry Pi Pico en MicroPython, vous aurez besoin de trois LED (rouge, jaune, verte) et de résistances appropriées pour les connecter (le composant reçu en classe est prêt à l'emploi). Voici un exemple de code qui montre comment cela peut être réalisé :

► Matériel nécessaire :

- Raspberry Pi Pico.
- Un composant "Led traffic lights"
- Câbles de connexion.

► Connexion des composants :

Connectez la LED rouge à une broche GPIO (par exemple GPIO2), la LED jaune à une autre broche (par exemple GPIO3) et la LED verte à une autre broche (par exemple GPIO4).

► Code MicroPython pour contrôler les LEDs :

```
import machine
import utime

# Configuration des broches pour chaque LED
led_rouge = machine.Pin(2, machine.Pin.OUT)
led_jaune = machine.Pin(3, machine.Pin.OUT)
led_verte = machine.Pin(4, machine.Pin.OUT)

def feu_rouge():
    led_rouge.on()
    led_jaune.off()
    led_verte.off()
    utime.sleep(5) # Rouge pour 5 secondes

def feu_jaune():
    led_rouge.off()
    led_jaune.on()
    led_verte.off()
    utime.sleep(2) # Jaune pour 2 secondes

def feu_vert():
    led_rouge.off()
    led_jaune.off()
    led_verte.on()
    utime.sleep(5) # Vert pour 5 secondes

while True:
    feu_rouge()
    feu_jaune()
    feu_vert()
```

► Explication du code :

- Les LED sont initialisées en tant que sorties de broches GPIO.
- Trois fonctions (`feu_rouge` , `feu_jaune` , `feu_vert`) sont définies pour contrôler les états des LED.
- Chaque fonction active une LED tout en éteignant les autres, simulant ainsi un feu de signalisation.
- Une boucle infinie (`while True`) alterne entre les trois états, imitant le comportement d'un feu de signalisation réel.

N'oubliez pas de tester soigneusement votre circuit et votre code dans un environnement sûr pour éviter tout court-circuit ou dommage au Raspberry Pi Pico.

Introduction aux buzzers

*Dans cet article, nous allons poser les bases nécessaires pour comprendre ce qu'est un buzzer, à quoi il sert et comment il est utilisé avec une carte microcontrôleur comme la Raspberry Pi Pico. Cet article est un ****prérequis conceptuel**** : il ne cherche pas encore à produire des sons complexes, mais à clarifier les différences fondamentales entre les types de buzzers et leurs implications côté matériel et côté code.*

Qu'est-ce qu'un buzzer ?

Un buzzer est un **composant électronique de sortie** dont le rôle est de produire un son.

On le rencontre très souvent dans des systèmes embarqués :

- bips de confirmation
- alertes sonores
- signaux d'erreur
- minuteries
- interfaces homme-machine simples

Contrairement à un haut-parleur, un buzzer est conçu pour être **simple à piloter**, avec peu de composants externes.

À quoi sert un buzzer dans un projet embarqué ?

Dans un projet pédagogique ou industriel, le buzzer sert principalement à :

- donner un **retour immédiat** à l'utilisateur
- signaler un **événement important** sans écran
- attirer l'attention (alarme, avertissement)

Il est particulièrement utile lorsque :

- l'affichage est absent ou minimal
- le système doit rester simple et robuste

Les deux grandes familles de buzzers

Il existe deux types de buzzers très courants en électronique embarquée :

- le **buzzer actif**
- le **buzzer passif**

Ils se ressemblent physiquement, mais leur fonctionnement interne et leur usage en programmation sont très différents.

Le buzzer actif

Un buzzer actif contient un **oscillateur interne**.

Cela signifie qu'il est capable de produire un son **tout seul**, dès qu'il est alimenté.

► Caractéristiques principales

- une seule tonalité possible
- fréquence interne fixe
- pilotage très simple
- aucun réglage de hauteur sonore

► Côté code

Pour faire sonner un buzzer actif :

- on met la broche à l'état HIGH → il sonne
- on met la broche à l'état LOW → il se tait

Il se pilote donc comme une **LED**, mais au lieu de s'allumer, il émet un son.

Le buzzer passif

Un buzzer passif **ne contient aucun oscillateur**.

Il ne peut pas produire de son sans un signal électrique approprié.

► Caractéristiques principales

- nécessite un signal PWM (= qui varie)
- la fréquence du signal détermine la hauteur du son
- le rapport cyclique influence le volume
- permet de produire des sons variés, des alertes et des mélodies

► Côté code

Avec un buzzer passif, le microcontrôleur doit :

- générer un signal périodique
- choisir la fréquence
- gérer la durée

Cela demande plus de code, mais offre beaucoup plus de possibilités.

Différences fondamentales entre actif et passif

Buzzer actif

- autonome
- simple
- une seule note
- peu flexible

Buzzer passif

- dépend du microcontrôleur
 - plus complexe
 - fréquences multiples
 - très flexible
-

Pourquoi a-t-on souvent l'impression que le buzzer actif sonne plus fort ?

C'est une impression fréquente, mais trompeuse.

Un buzzer actif :

- fonctionne directement à sa fréquence de résonance
- est souvent alimenté avec une tension stable
- est optimisé pour produire un son immédiatement audible

Un buzzer passif :

- est souvent utilisé avec des réglages prudents
- est piloté en basse tension
- dépend fortement du signal généré par le code

Un buzzer passif **peut sonner aussi fort, voire plus fort**, mais uniquement s'il est correctement piloté (fréquence, tension, duty cycle).

Choisir le bon buzzer selon le projet

Utiliser un **buzzer actif** si :

- un simple bip suffit
- le code doit rester très simple
- on veut juste un signal sonore

Utiliser un **buzzer passif** si :

- on veut contrôler la hauteur du son
 - produire des alertes complexes
 - jouer des mélodies
 - expérimenter avec le PWM
-

Erreurs fréquentes

- Confondre buzzer actif et passif
 - Essayer de changer la fréquence d'un buzzer actif
 - Utiliser un buzzer passif sans PWM
 - Conclure que le buzzer passif est "moins puissant" par nature
-

Ce qu'il faut retenir

- Un buzzer est un dispositif de sortie sonore simple
 - Actif et passif ont des usages très différents
 - Le type de buzzer détermine directement le **type de code**
 - La complexité sonore vient du **pilotage**, pas du composant seul
-

Dans le prochain article, nous utiliserons un **buzzer actif** pour produire des bips, gérer le rythme et construire progressivement un signal de type Morse, uniquement à l'aide de durées et de pauses.

Utiliser un buzzer actif (KY-012)

*Dans cet article, nous allons apprendre à utiliser un **buzzer actif KY-012** avec une Raspberry Pi Pico. L'objectif est de comprendre comment produire des bips sonores, gérer le rythme et utiliser le temps comme information, jusqu'à construire un **code Morse simplifié**.*

Contrairement au buzzer passif, ici **la fréquence n'est pas contrôlable** : le buzzer actif ne sait produire **qu'une seule note**.

Rappel : principe du buzzer actif

Un buzzer actif contient un oscillateur interne. Dès qu'il est alimenté, il émet un son à une fréquence fixe.

Conséquence directe

- Pas de PWM
- Pas de fréquence à choisir
- Uniquement ON / OFF

Côté code, on travaille uniquement avec :

- l'état de la broche
- la durée pendant laquelle elle reste active

Connexion du KY-012

- **S** (Signal) → GPIO15
- **-** (GND) → GND
- **+** (VCC) → 3.3 V

Contrairement au buzzer passif, le buzzer actif **doit être alimenté**.

Premier son : un bip simple

On commence par le cas le plus élémentaire : un bip, puis arrêt.

```
from machine import Pin
import utime

buzzer = Pin(15, Pin.OUT)

buzzer.value(1)      # buzzer ON
utime.sleep(1)        # 1 seconde
buzzer.value(0)      # buzzer OFF
```

► À retenir

- `value(1)` alimente le buzzer
- `value(0)` coupe le son
- la durée dépend uniquement du `sleep`

Bip + pause en boucle

On introduit maintenant la répétition.

```
from machine import Pin
import utime

buzzer = Pin(15, Pin.OUT)

while True:
    buzzer.value(1)
    utime.sleep(0.5)

    buzzer.value(0)
    utime.sleep(0.5)
```

Le buzzer émet un bip régulier, comparable à un métronome.

Bip court et bip long

La différence entre un bip court et un bip long est **uniquement temporelle**.

```
from machine import Pin
import utime

buzzer = Pin(15, Pin.OUT)

while True:
    # bip court
    buzzer.value(1)
    utime.sleep(0.2)
    buzzer.value(0)
    utime.sleep(0.3)

    # bip long
    buzzer.value(1)
    utime.sleep(0.6)
    buzzer.value(0)
    utime.sleep(0.8)
```

Introduire la notion de rythme

À ce stade, on ne change jamais le son lui-même. On ne joue que sur :

- la durée du bip
- la durée du silence

Avec un buzzer actif, **le rythme est l'unique information sonore**.

C'est exactement le principe du code Morse.

Principe du Morse (version pédagogique)

Le code Morse repose sur :

- un son court
- un son long
- des pauses

Pour simplifier :

- point → bip court
- trait → bip long

Nous n'utilisons pas ici les durées normalisées strictes, mais une version **compréhensible et lisible**.

Dictionnaire Morse simplifié

On associe chaque lettre à une séquence de symboles.

```
morse = {  
    'S': '...',  
    'O': '---',  
    'I': '..',  
    'N': '-.',  
    'F': '..-.'  
}
```

Jouer "SOS"

```
from machine import Pin  
import utime  
  
buzzer = Pin(15, Pin.OUT)  
  
morse = {  
    'S': '...',  
    'O': '---'  
}  
  
dot = 0.2  
dash = 0.6  
pause = 0.3  
  
while True:  
    for letter in "SOS":  
        for symbol in morse[letter]:  
            buzzer.value(1)  
            if symbol == '.':  
                utime.sleep(dot)  
            else:  
                utime.sleep(dash)  
  
            buzzer.value(0)  
            utime.sleep(pause)  
  
        utime.sleep(0.6)    # pause entre lettres
```

Jouer "INFO"

```
morse = {  
    'I': '...',  
    'N': '-.',  
    'F': '..-.',  
    'O': '----'  
}  
  
message = "INFO"
```

Le reste du code est identique : seul le dictionnaire et le message changent.

Ce que cet article a introduit

- pilotage ON / OFF
- gestion du temps
- notion de rythme
- dictionnaire comme table de correspondance
- boucle imbriquée

Erreurs fréquentes

- Essayer de modifier la fréquence d'un buzzer actif
- Oublier la pause entre lettres
- Alimenter le buzzer actif sans GND commun

Ce qu'il faut retenir

- Un buzzer actif ne produit qu'un seul son
- Toute l'information passe par la durée
- Le Morse est une application directe de cette logique
- Simplicité matérielle ≠ pauvreté pédagogique

Dans le prochain article, nous utiliserons un **buzzer passif (KY-006)** pour introduire la fréquence, le volume et la génération de sons évolutifs, jusqu'à des mélodies complètes.

Utiliser un buzzer passif (KY-006)

Dans cet article, nous passons au **buzzer passif KY-006**, beaucoup plus flexible que le buzzer actif. Il permet de **contrôler la hauteur du son**, le volume et les variations dans le temps. C'est avec lui que l'on peut produire des **alertes évoluées** et, progressivement, de **véritables mélodies**.

Principe du buzzer passif

Un buzzer passif **ne produit aucun son tout seul**. Il doit être piloté par un **signal PWM**.

Avec un buzzer passif :

- la **fréquence** du PWM détermine la hauteur du son
- le **duty cycle** influence le volume
- la **durée** définit le rythme

On contrôle donc **trois paramètres**, contre un seul pour le buzzer actif.

Connexion du KY-006

- **S** (Signal) → GPIO15 (PWM)
- **-** (GND) → GND
- **+** → non connecté (c'est le **S** qui sert à générer le signal dans notre cas)

Initialisation de base

```
from machine import Pin, PWM
import utime

buzzer = PWM(Pin(15))
buzzer.duty_u16(2000)
```

À ce stade, le buzzer est prêt, mais **silencieux** tant qu'aucune fréquence n'est définie.

Alerte simple sur deux tons (ambulance)

On alterne deux fréquences distinctes.

```
while True:
    buzzer.freq(600)
    utime.sleep(0.5)

    buzzer.freq(900)
    utime.sleep(0.5)
```

Alerte avec montée progressive de fréquence

```
while True:
    for freq in range(400, 1000, 20):
        buzzer.freq(freq)
        utime.sleep(0.02)
```

La fréquence augmente progressivement, ce qui donne une sensation de montée de tension.

Alerte avec descente progressive de fréquence

```
while True:
    for freq in range(1000, 400, -20):
        buzzer.freq(freq)
        utime.sleep(0.02)
```

Montée puis descente de fréquence

```
while True:
    for freq in range(400, 1000, 20):
        buzzer.freq(freq)
        utime.sleep(0.02)

    for freq in range(1000, 400, -20):
        buzzer.freq(freq)
        utime.sleep(0.02)
```

Alerte à deux tons avec durée qui diminue

La durée de chaque note diminue de **25 % à chaque itération**.

```
duration = 2.0

while duration > 0.3:
    buzzer.freq(600)
    utime.sleep(duration)

    buzzer.freq(900)
    utime.sleep(duration)

    duration = duration * 0.75
```

Diminuer aussi la pause

```
duration = 2.0
pause = 1.0

while duration > 0.3:
    buzzer.freq(600)
    utime.sleep(duration)
    utime.sleep(pause)

    buzzer.freq(900)
    utime.sleep(duration)
    utime.sleep(pause)

    duration = duration * 0.75
    pause = pause * 0.75
```

Introduction aux notes de musique

On peut associer des noms de notes à des fréquences.

```
notes = {  
    'C': 262,  
    'D': 294,  
    'E': 330,  
    'F': 349,  
    'G': 392,  
    'A': 440,  
    'B': 494,  
    'C5': 523,  
    'D5': 587,  
    'E5': 659,  
    'F5': 698,  
    'G5': 784,  
    'A5': 880  
}
```

Mélodie simple, rythme monotone

```
melody = ['C', 'D', 'E', 'F', 'G', 'A', 'G', 'F']  
  
for note in melody:  
    buzzer.freq(notes[note])  
    utime.sleep(0.4)
```

Introduire la durée avec des tuples

Chaque élément contient :

- la note
- la durée

```
melody = [  
    ('C', 0.4),  
    ('D', 0.4),  
    ('E', 0.6),  
    ('C', 0.6)  
]
```

Lecture :

```
for note, duration in melody:
    buzzer.freq(notes[note])
    utime.sleep(duration)
```

Frère Jacques (8 premières notes)

```
melody = [
    ('C', 0.4), ('D', 0.4), ('E', 0.4), ('C', 0.4),
    ('C', 0.4), ('D', 0.4), ('E', 0.4), ('C', 0.4)
]

for note, duration in melody:
    buzzer.freq(notes[note])
    utime.sleep(duration)
```

Joyeux anniversaire (6 premières notes)

```
melody = [
    ('G', 0.4), ('G', 0.2),
    ('A', 0.6),
    ('G', 0.6),
    ('C5', 0.6),
    ('B', 1.0)
]

for note, duration in melody:
    buzzer.freq(notes[note])
    utime.sleep(duration)
```

Le finale

```
melody = [  
    ('A4', 0.3), ('B4', 0.3), ('D5', 0.4), ('B4', 0.4),  
    ('F5', 0.6), ('F5', 0.6),  
    ('E5', 0.8)  
]  
  
notes.update({  
    'A4': 440,  
    'B4': 494,  
    'D5': 587,  
    'E5': 659,  
    'F5': 698  
})  
  
for note, duration in melody:  
    buzzer.freq(notes[note])  
    utime.sleep(duration)
```

Erreurs fréquentes

- oublier d'utiliser le PWM
- confondre fréquence et durée
- jouer trop fort dès le début
- croire qu'une mélodie est autre chose qu'une suite de fréquences et de temps

Ce qu'il faut retenir

- le buzzer passif est entièrement piloté par le code
- fréquence = hauteur du son
- durée = rythme
- les mélodies sont des données, pas de la magie
- une structure simple permet des résultats complexes

Avec ces trois articles, les élèves disposent désormais :

- d'une vision claire des buzzers
- d'une progression logique
- d'un terrain idéal pour introduire abstraction, fonctions... et surprises sonores ultérieures

Les capteurs

*Un capteur est un dispositif qui **détecte et mesure une grandeur physique ou une caractéristique de son environnement**, puis **convertit** cette information en un signal utilisable par un système électronique. Les capteurs jouent un rôle essentiel en robotique, car **ils permettent aux robots de percevoir** et d'interagir avec leur environnement.*

Utilité des capteurs

Les capteurs sont utilisés pour fournir aux robots des informations sur le monde qui les entoure. Ils permettent au robot de **recueillir des données sur des grandeurs physiques** telles que la lumière, le son, la température, la pression, la distance, etc. Ces informations sont ensuite utilisées **pour prendre des décisions**, effectuer des actions et interagir avec l'environnement de manière intelligente.

Les capteurs permettent aux robots de **détecter des objets**, d'analyser des caractéristiques de l'environnement, de suivre des lignes, d'éviter des obstacles, de mesurer des paramètres, de percevoir des mouvements, de reconnaître des formes et de nombreux autres aspects nécessaires à leur fonctionnement autonome.

Différents types de capteurs

On distingue généralement deux types principaux de capteurs en fonction de la manière dont ils fournissent des informations : les **capteurs analogiques** et les **capteurs numériques**.

- **Capteurs analogiques** : Les capteurs analogiques fournissent une sortie continue qui varie en fonction de la grandeur physique mesurée. Ils produisent **une tension ou un courant proportionnel à la grandeur mesurée**. Les valeurs de sortie varient généralement de manière linéaire ou non linéaire en fonction de la grandeur mesurée. Les signaux analogiques nécessitent souvent une conversion en signaux numériques pour être traités par un système électronique.
- **Capteurs numériques** : Les capteurs numériques fournissent une sortie discrète, généralement sous forme de bits (0 et 1) ou de valeurs numériques prédéfinies. Ils utilisent des composants électroniques internes pour convertir la grandeur physique en signaux numériques. Les capteurs numériques sont souvent plus précis, plus faciles à utiliser et peuvent être directement connectés à des microcontrôleurs ou à d'autres systèmes électroniques.

Il existe une grande variété de capteurs disponibles, chacun étant conçu pour mesurer une grandeur spécifique. Parmi les autres types de capteurs couramment utilisés en robotique, on peut citer les capteurs de mouvement, les capteurs de proximité, les capteurs de gaz, les capteurs de pression, les capteurs de température, les capteurs de lumière, les capteurs de force, les capteurs de couleur, les capteurs de son, les capteurs d'inclinaison, etc.

L'utilisation appropriée des capteurs permet aux robots de percevoir leur environnement, d'interagir de manière autonome et d'exécuter des tâches spécifiques en fonction des informations recueillies.

Principaux capteurs utilisés en robotique

Voici une liste des principaux capteurs utilisés en robotique, accompagnée d'une brève description de chacun :

Capteur de distance ultrasonique : Ce capteur utilise des ondes sonores pour mesurer la distance entre le robot et un objet environnant. Il est largement utilisé pour l'évitement d'obstacles et la navigation.

En classe, nous avons utilisé le [capteur ultra-son HC-SR04](#), qui est un modèle très répandu et bon marché.

Capteur infrarouge : Les capteurs infrarouges détectent les rayonnements infrarouges émis par les objets. Ils sont utilisés pour la détection d'obstacles, le suivi de lignes, la détection de mouvement, etc.

Capteur de lumière : Les capteurs de lumière mesurent l'intensité de la lumière ambiante. Ils peuvent être utilisés pour détecter la luminosité de l'environnement et ajuster le comportement du robot en conséquence.

Capteur de température : Ce capteur mesure la température de l'environnement. Il est utilisé dans diverses applications telles que la surveillance de la température, la régulation thermique et le contrôle climatique.

En classe, nous avons utilisé le [capteur de température et d'humidité DHT11](#), qui est un modèle très répandu et bon marché.

Capteur de pression : Les capteurs de pression mesurent la force appliquée sur une surface. Ils sont utilisés dans les applications de contrôle de la force, de détection de contact et de mesure de la pression atmosphérique.

Capteur de mouvement : Ces capteurs détectent les mouvements, les inclinaisons ou les rotations du robot. Ils sont couramment utilisés pour la détection de mouvement, la stabilisation et le contrôle de l'orientation.

Capteur de son : Les capteurs de son détectent les variations de pression acoustique dans l'environnement. Ils peuvent être utilisés pour détecter des sons spécifiques, effectuer une reconnaissance vocale ou analyser l'acoustique de l'environnement.

Capteur de couleur : Ces capteurs permettent de détecter les couleurs des objets. Ils sont utilisés dans les applications de tri, d'identification d'objets et de suivi de lignes basé sur la couleur.

Capteur de proximité : Les capteurs de proximité détectent la présence d'objets à proximité sans contact physique direct. Ils sont utilisés pour la détection d'obstacles, la détection de présence et l'activation de fonctions basées sur la proximité.

Capteur de gaz : Ces capteurs détectent la présence et la concentration de gaz dans l'environnement. Ils sont utilisés dans les applications de surveillance de la qualité de l'air, de détection de fuites de gaz, etc.

Oui, il existe d'autres capteurs utilisés en robotique. Voici quelques exemples supplémentaires :

Capteur d'inclinaison : Ces capteurs mesurent l'inclinaison ou l'orientation du robot par rapport à la gravité. Ils sont utilisés pour la stabilisation, la détection de l'attitude et le contrôle de l'équilibre.

Capteur de force : Les capteurs de force mesurent la force appliquée sur une surface ou un objet. Ils sont utilisés dans les applications de manipulation d'objets, de contrôle de la pression et de rétroaction haptique.

Capteur d'humidité : Ces capteurs mesurent le niveau d'humidité dans l'air ou dans un matériau. Ils sont utilisés dans les applications de contrôle de l'humidité, d'irrigation intelligente et de surveillance environnementale.

Capteur de gaz toxique : Ces capteurs spécifiques détectent la présence de gaz toxiques ou dangereux dans l'environnement. Ils sont utilisés dans les applications de sécurité industrielle, de surveillance environnementale et de détection de fuites.

Capteur de mouvement oculaire : Ces capteurs utilisent la technologie de suivi oculaire pour détecter et mesurer les mouvements des yeux. Ils sont utilisés dans les interfaces homme-machine avancées et les applications de réalité virtuelle.

Capteur d'odeur : Les capteurs d'odeur, également appelés capteurs de gaz odorants, peuvent détecter et analyser différentes odeurs. Ils sont utilisés dans des applications telles que l'analyse de la qualité de l'air, la détection d'odeurs spécifiques et le contrôle des émissions odorantes.

Capteur de vitesse : Ces capteurs mesurent la vitesse de déplacement du robot. Ils sont utilisés pour le contrôle de la vitesse, la détection de mouvement rapide et le suivi de la trajectoire.

Capteur de vision : Les capteurs de vision, tels que les caméras haute résolution, sont utilisés pour capturer des images et des vidéos de l'environnement du robot. Ils sont essentiels pour la perception visuelle, la détection d'objets, la reconnaissance faciale, etc.

Chaque capteur a des caractéristiques spécifiques et est utilisé pour des applications particulières en fonction des besoins du robot et de son environnement. L'utilisation de différents capteurs permet d'obtenir une perception complète et précise du monde extérieur, ce qui est essentiel pour des opérations robotiques avancées. Ces capteurs sont largement utilisés en robotique et offrent une multitude de possibilités pour la perception et l'interaction du robot avec son environnement. En fonction des besoins spécifiques de chaque projet, il est possible d'utiliser différents capteurs pour obtenir les informations nécessaires et permettre au robot d'interagir de manière intelligente avec le monde qui l'entoure.

BME280: pression atmosphérique et altitude

Dans cette activité, tu vas découvrir comment mesurer la pression atmosphérique avec le capteur BME280 et utiliser ces données pour calculer l'altitude, une compétence essentielle pour comprendre et analyser la descente d'un CanSat lors de son vol.

Objectif du cours

Dans ce cours, tu vas apprendre à :

- lire la **pression atmosphérique** avec le capteur **BME280** via `raw_values` ,
- comprendre la relation entre pression et altitude,
- calculer l'**altitude** en utilisant une formule standard.

Ces compétences te serviront pour la **mission primaire** du CanSat.

Comprendre la pression atmosphérique

▶ Qu'est-ce que la pression atmosphérique ?

La pression atmosphérique, c'est le **poids de l'air** au-dessus de toi.

- Au niveau de la mer : environ **1013 hPa**
- En altitude, la pression **diminue** car la colonne d'air est plus petite.

👉 Au CanSat, la pression va donc diminuer beaucoup *pendant la montée* puis augmenter pendant la descente.

Le capteur BME280

▶ Mesures disponibles

- Température (°C)
- Pression (hPa)
- Humidité (%)

▶ Communication

Il utilise le **bus I2C**, qui ne demande que deux fils : SDA et SCL.

► Exemple de branchement (Pico → BME280)

BME280	PICO
VCC	3.3V
GND	GND
SCL	GP9
SDA	GP8

Lire les données *numériques* avec `raw_values`

Le module BME280 utilisé dans le CanSat propose :

```
temp, pressure, humidity = bme.raw_values
```

Il retourne :

- la température en **°C**,
- la pression en **hPa**,
- l'humidité en %.

👉 Pas besoin de convertir des chaînes : c'est prêt pour les calculs.

► Exemple de code

```
from machine import I2C, Pin
from bme280 import BME280
import time

i2c = I2C(0, scl=Pin(9), sda=Pin(8))
bme = BME280(i2c=i2c)

while True:
    temp, pressure, humidity = bme.raw_values
    print("Température (°C):", temp)
    print("Pression (hPa):", pressure)
    print("Humidité (%)ate", humidity)
    print("-----")
    time.sleep(1)
```



Comprendre le calcul de l'altitude à partir de la pression

Pour estimer l'altitude à partir de la pression mesurée par le BME280, on utilise une formule barométrique simplifiée. Mais avant de l'utiliser, il faut comprendre un point essentiel : **la pression au niveau de la mer n'est pas une constante !**

► Qu'est-ce que la pression MSL ?

MSL signifie **Mean Sea Level pressure** = pression ramenée au niveau de la mer. C'est la valeur que les stations météo publient dans les bulletins.

IMPORTANT :

- La valeur "standard" est **1013.25 hPa**, mais elle varie **chaque jour**, selon la météo.
- Lors d'un **anticyclone**, la pression peut monter jusqu'à **1035 hPa**.
- Lors d'une **dépression**, elle peut descendre vers **980 hPa** (ou moins).

👉 Si tu utilises la valeur **standard 1013.25 hPa** alors que la pression réelle du jour est par exemple 1002 hPa, ton altitude sera fausse de plusieurs **dizaines de mètres**.

► Où trouver la pression MSL du jour ?

Voici des sources fiables (utiliser la station météo la plus proche) :

- **IRM – Belgique** <https://www.meteo.be> Chercher "pression atmosphérique" → valeur en hPa.
- **OpenWeatherMap** (API ou application) <https://openweathermap.org>
- **Météociel** <https://www.meteociel.fr/observations-meteo/pression.php>
- **Applications smartphone** : Météo & Radar, Windy, etc.

👉 Dans un projet CanSat, on choisit la station météo la plus proche du lieu du **lancement**.

► La formule utilisée

La relation pression ↔ altitude vient de la physique de l'atmosphère :

$$\text{altitude} = 44330 \times \left(1 - \left(\frac{P}{P_0} \right)^{\frac{1}{5.255}} \right)$$

- **P** = pression mesurée par le BME280 (en hPa)
- **P₀** = pression MSL (pression au niveau de la mer, en hPa)
- **altitude** en mètres

POURQUOI ÇA FONCTIONNE ?

L'air est comprimé par son propre poids. Plus on monte :

- la densité de l'air diminue,

- la pression diminue.

La formule ci-dessus exprime précisément cette diminution.

► Exemple concret

Imaginons que tu mesures :

- Pression du capteur : **955 hPa**
- Pression MSL du jour (IRM) : **1002 hPa**

$$\text{altitude} = 44330 \times \left(1 - (955/1002)^{1/5.255}\right)$$

→ Résultat $\approx$ **423 m**

Si tu avais utilisé 1013.25 hPa par défaut :

→ $\approx$ **500 m** (erreur de +77 m)

► Pourquoi la pression MSL change ?

Parce que la pression dépend de la masse d'air au-dessus de la région :

- Air chaud → plus léger → **basse pression**
- Air froid → plus lourd → **haute pression**
- Mouvements atmosphériques → variations permanentes

Donc, **tu dois calibrer ton capteur** en mettant à jour le **P0** **chaque jour** ou avant chaque vol.

Faut-il utiliser la pression MSL du lieu ?

Oui, mais il faut bien comprendre ce que cela signifie.

► 1) La station météo te donne la pression ramenée au niveau de la mer

Quand tu vas sur IRM, OpenWeather, Windy, etc., la pression affichée n'est **pas** la pression réellement mesurée à cet endroit. Les stations appliquent un calcul pour "ramener" la pression mesurée au **niveau de la mer (MSL)** afin de pouvoir comparer deux stations situées à des altitudes différentes.

Exemple :

- Pression mesurée à Bastogne (500 m) $\approx$ 955 hPa
- Pression ramenée au niveau de la mer (MSL) $\approx$ 1010 hPa

👉 C'est cette **pression MSL** (1010 hPa) qu'il faut utiliser dans la formule d'altitude.

Pourquoi ? Parce que la formule barométrique **suppose que P₀ correspond à la pression au niveau de la mer**, pas à la pression "au sol" à 350 ou 500 m d'altitude.

◆ Alors quelle valeur utiliser dans le CanSat ?

▶ ✓ Option 1 (recommandée par l'ESA & IRM)

➡ P_0 = pression MSL du jour fournie par la station météo la plus proche du site de lancement.

Sources :

- IRM Belgique : <https://www.meteo.be>
- Windy : <https://www.windy.com>
- OpenWeatherMap
- Stations météo universitaires locales

C'est la méthode "officielle".

▶ ◆ Option 2 : calibration locale pour plus de précision

Les conditions météo changent en permanence. Même la valeur MSL donnée par IRM peut entraîner une erreur (surtout si la station est loin).

👉 Solution : **calibrer P_0 sur place**.

ÉTAPES :

1. Tu regardes la **vraie altitude** du site (Google Maps, géoportail, panneau local).
2. Tu mesures la pression du BME280 sur place : **P_{mesure}** .
3. Tu ajustes P_0 pour que la formule te redonne cette altitude.

EXEMPLE

Lieu du lancement : altitude réelle = **510 m** Pression mesurée BME280 = **955.4 hPa**

Tu cherches la valeur P_0 qui donne 510 m :

[$510 = 44330 \times \left(1 - (955.4/P_0)^{1/5.255}\right)$]

Résultat → **$P_0 \approx 1010.6 \text{ hPa}$**

👉 Tu mets **$P_0 = 1010.6$** dans ton code.

C'est la méthode la **plus précise** possible.

◆ 5) Mini-algorithme conseillé pour le CanSat

1. Récupérer la pression MSL du jour (IRM).
2. Mesurer la pression réelle au sol avec le BME280.
3. Calculer l'altitude affichée → comparer avec l'altitude connue.
4. **Ajuster P_0 automatiquement** dans le code pour corriger l'erreur.
5. Laisser P_0 constant pendant tout le vol.

Calculer l'altitude à partir de la pression

On utilise la formule barométrique simplifiée :

$$\text{altitude} = 44330 \times \left(1 - \left(\frac{P}{P_0} \right)^{\frac{1}{5.255}} \right)$$

- P : pression mesurée (hPa)
- P_0 : pression au niveau de la mer (par défaut 1013.25 hPa)

► Code "complet" avec altitude

```
from machine import I2C, Pin
from bme280 import BME280
import time

i2c = I2C(0, scl=Pin(9), sda=Pin(8))
bme = BME280(i2c=i2c)

P0 = 1013.25  # pression standard au niveau de la mer

while True:
    temp, P, hum = bme.raw_values

    # ICI: CALCULE L'ALTITUDE AVEC LA FORMULE SIMPLIFIEE

    print(f"Température: {temp:.2f} °C")
    print(f"Pression: {P:.2f} hPa")
    print(f"Altitude estimée: {altitude:.2f} m")
    print("-----")

    time.sleep(1)
```

Exemple d'interprétation

PRESSION MESURÉE (HPA)	ALTITUDE ESTIMÉE
1013	0 m
1000	~110 m
950	~520 m
900	~980 m
800	~1900 m

👉 Pendant le vol du CanSat, tu vas observer une **chute rapide de pression**, ce qui te permet de **recalculer l'altitude en temps réel**.

À retenir

- Le BME280 renvoie des **valeurs numériques** grâce à `raw_values`.
- La pression diminue quand l'altitude augmente.
- On peut calculer l'altitude **en direct** pendant le vol.
- Ces données sont la base de la **mission primaire** du CanSat.



Exercice

1. Lire les valeurs du capteur avec `raw_values`.
2. Afficher température + pression + altitude chaque seconde.
3. Monter un étage avec le capteur et observer comment l'altitude change.
4. Tester à l'extérieur : comparer avec l'altitude donnée par Google Maps.

DHT11: Simulation du Capteur

*Tu vas apprendre à utiliser un ****objet Python**** qui simule un capteur de température et d'humidité (le DHT11). Cela te permettra d'écrire du code ****exactement comme si tu avais un vrai capteur branché**** sur un Raspberry Pi Pico.*

Utiliser la classe `DHT11` en Python

Tu dois d'abord télécharger le fichier Python ci-joint. Ensuite, crée un nouveau fichier python dans Thonny que tu appelles "simulation_dht11.py" et sauvegarde-le dans ton dossier "Téléchargements".

Étape 1 : importer le module

Quand on veut utiliser une classe qui se trouve dans un autre fichier Python, on doit **l'importer**.

👉 Imaginons que ton fichier avec la classe `DHT11` s'appelle `dht11.py`. Alors, tu dois écrire en haut de ton programme :

```
from dht11 import DHT11
```

Étape 2 : créer un objet capteur

Maintenant, tu peux **créer un capteur simulé**. Il faut lui donner le numéro de la broche (pin) sur laquelle il est censé être branché :

```
capteur = DHT11(pin=15)
```

Même si la simulation n'utilise pas vraiment la broche, on garde ce paramètre pour ressembler au vrai capteur.

Étape 3 : demander une mesure

Avant de lire la température ou l'humidité, tu dois **lancer une mesure** :

```
capteur.measure()
```

C'est comme si tu envoyais l'ordre au vrai capteur de "prendre une photo" de l'air ambiant.

Étape 4 : lire les valeurs

Une fois la mesure faite, tu peux demander :

```
print("Température :", capteur.temperature(), "°C")
print("Humidité :", capteur.humidity(), "%")
```

Chaque appel va te donner la **dernière valeur mesurée**.

Exemple complet

```
from dht11 import DHT11

# Création d'un capteur simulé
capteur = DHT11(pin=15)

# Faire 3 mesures
for i in range(3):
    capteur.measure() # on lance une nouvelle mesure
    print("Température :", capteur.temperature(), "°C")
    print("Humidité :", capteur.humidity(), "%")
    print("---")
```

À retenir

1. On **importe** la classe depuis son fichier.
2. On **crée un objet** `DHT11`.
3. On appelle `measure()` pour lancer une mesure.
4. On récupère les valeurs avec `temperature()` et `humidity()`.

👉 Quand tu passeras au vrai Raspberry Pi Pico, tu n'auras presque rien à changer : seule la ligne `from dht11 import DHT11` sera remplacée par l'import du vrai module.

Veux-tu que je t'écrive aussi la **version avec le vrai module** `dht` de MicroPython pour montrer le passage simulation → réel ?

DHT11: Capteur de température et d'humidité

Cet article présente le DHT11, un capteur de température et d'humidité très répandu et facile à utiliser.

- Le capteur DHT11 est un capteur d'humidité et de température numérique économique et couramment utilisé.
- Il fournit des lectures de température et d'humidité relativement précises à partir d'une seule broche de données.



Caractéristiques

Le capteur DHT11 est un capteur de température et d'humidité très couramment utilisé dans de nombreuses applications. Voici quelques-unes de ses caractéristiques principales :

1. **Mesure de la température et de l'humidité** : Le capteur DHT11 permet de mesurer à la fois la température et l'humidité relative de l'environnement dans lequel il est placé.
2. **Précision** : Le capteur DHT11 offre une **précision de $\pm 2^{\circ}\text{C}$** pour la mesure de la température et de **$\pm 5\%$** pour la mesure de l'humidité relative.
3. **Plage de mesure** : La plage de mesure de la température du capteur DHT11 est généralement de **0°C à 50°C** , tandis que la plage de mesure de l'humidité relative est de **20% à 90%**.
4. **Alimentation** : Le capteur DHT11 fonctionne avec une tension d'alimentation de **3,3V à 5V**, ce qui le rend compatible avec de nombreux microcontrôleurs et cartes de développement.
5. **Sortie numérique** : Le capteur DHT11 utilise une **sortie numérique** pour transmettre les données de température et d'humidité. Il envoie les informations sous forme de signal série numérique.
6. **Temps de réponse** : Le capteur DHT11 a un **temps de réponse relativement lent** par rapport à d'autres capteurs de température et d'humidité. Il peut prendre plusieurs secondes pour effectuer une mesure et transmettre les données.
7. **Connectivité** : Le capteur DHT11 utilise un protocole de communication à **un seul fil**, ce qui le rend **facile à intégrer** dans des projets électroniques et des systèmes embarqués.

Il convient de noter que le capteur DHT11 est un **capteur d'entrée de gamme** largement utilisé pour des applications simples de mesure de température et d'humidité. Si une précision plus élevée ou une plus grande plage de mesure est requise, il existe d'autres capteurs plus avancés disponibles sur le marché (tels que le DHT22 ou le BME280)

DHT22: alternative plus précise

Parmi les modèles plus "haut de gamme", on citera par exemple le **DHT22**. Également connu sous le nom d'AM2302, il offre une **meilleure précision** et une **plus large plage de mesure** par rapport au DHT11. Il peut mesurer la température avec une précision de $\pm 0,5^{\circ}\text{C}$ et l'humidité avec une précision de $\pm 2\%$. De plus, le DHT22 a une plage de mesure de température de **-40°C à 80°C** et une plage d'humidité de **0% à 100%**.

Configuration matérielle

- Connectez la broche de données du capteur DHT11 au GPIO du Raspberry Pi Pico (par exemple, GPIO2).
- Alimentez le capteur DHT11 en connectant le fil rouge au 3,3 V du Pico et le fil noir à la masse (GND).

Installation de la bibliothèque DHT11

- Téléchargez la bibliothèque dht11 pour MicroPython à partir du dépôt GitHub officiel (<https://github.com/mcauser/micropython-dht11>).
- Copiez le fichier `dht11.py` sur votre Raspberry Pi Pico.

Exemple de code pour la lecture des données du capteur DHT11

```
import machine
import dht
import time

# Initialize the DHT sensor.
capteur = dht.DHT11(machine.Pin(2))

while True:
    capteur.measure() # Trigger a measurement.
    print("Température:", capteur.temperature())
    print("Humidité:", capteur.humidity())
    time.sleep(2)
```

Explication du code

Bien sûr, voici une explication pour ce code :

- `import machine, import dht, import time` : Ces trois instructions importent les modules nécessaires. `machine` est un module qui permet d'interagir avec le matériel de l'appareil, `dht` est un module pour travailler avec le capteur de température et d'humidité DHT11, et `time` est un module pour manipuler le temps (par exemple, pour créer des délais).
- `capteur = dht.DHT11(machine.Pin(2))` : Cette ligne crée une instance de l'objet DHT11 appelée `capteur`. Le DHT11 est connecté au pin 2 de la machine (en supposant que vous utilisiez un ESP8266 ou un ESP32, où le pin 2 est généralement disponible).
- `while True:` : Ceci commence une boucle infinie. Le code à l'intérieur de cette boucle continuera à s'exécuter indéfiniment.
- `capteur.measure()` : Cette ligne indique au capteur DHT11 de commencer une mesure. Le capteur effectue une lecture de la température et de l'humidité de l'air.
- `print("Température:", capteur.temperature())` : Cette ligne affiche la température mesurée par le capteur. La méthode `temperature()` renvoie la température mesurée lors du dernier appel à `capteur.measure()`.
- `print("Humidité:", capteur.humidity())` : De même, cette ligne affiche l'humidité mesurée par le capteur. La méthode `humidity()` renvoie l'humidité mesurée lors du dernier appel à `capteur.measure()`.
- `time.sleep(2)` : Pour finir, cette ligne fait une pause dans l'exécution du programme pendant 2 secondes. C'est important car le capteur DHT11 a besoin d'un certain temps entre les lectures pour garantir des mesures précises. En général, il est recommandé d'attendre au moins 2 secondes entre chaque lecture.

Donc, en résumé, ce code effectue une lecture continue de la température et de l'humidité avec un capteur DHT11, et imprime ces valeurs toutes les 2 secondes. Ressources supplémentaires

Infos supplémentaires

- Tutoriel vidéo sur l'utilisation du capteur DHT11 avec MicroPython et Raspberry Pi Pico : <https://youtu.be/hZ9zdvTRnTw>
- Documentation de la bibliothèque DHT11 pour MicroPython : <https://github.com/mcauser/micropython-dht11>

Ces instructions vous permettront de lire les données de température et d'humidité à partir du capteur DHT11 à l'aide de MicroPython sur votre Raspberry Pi Pico.

HC-SR04: Capteur de distance à ultra-sons

Cet article présente HC-SR04, un capteur de distance à ultra-sons très répandu et facile à utiliser.

Introduction au capteur de distance HC-SR04

- Le capteur de distance HC-SR04 est un capteur ultrasonique populaire utilisé pour mesurer la distance entre le capteur et un objet.
- Il utilise des ondes sonores ultrasoniques pour déterminer la distance, en émettant un signal et en mesurant le temps nécessaire pour que l'écho revienne.

Configuration matérielle

- Connectez la broche de déclenchement (Trig) du capteur HC-SR04 à un GPIO du Raspberry Pi Pico (par exemple, GPIO2).
- Connectez la broche d'écho (Echo) du capteur HC-SR04 à un autre GPIO du Pico (par exemple, GPIO3).
- Alimentez le capteur HC-SR04 en connectant le fil rouge au 5V du Pico et le fil noir à la masse (GND).

Exemple de code pour la mesure de distance avec le capteur HC-SR04

```
import machine
import utime

# Définir Les broches GPIO pour Le déclenchement et L'écho
trigger_pin = machine.Pin(2, machine.Pin.OUT)
echo_pin = machine.Pin(3, machine.Pin.IN)

# Fonction pour mesurer la distance
def mesure_distance():
    # Générer une impulsion de déclenchement
    trigger_pin.low()
    utime.sleep_us(2)
    trigger_pin.high()
    utime.sleep_us(10)
    trigger_pin.low()

    # Attendre Le signal d'écho
    while echo_pin.value() == 0:
        signal_depart = utime.ticks_us()

    while echo_pin.value() == 1:
        signal_arrivee = utime.ticks_us()

    # Calculer La durée de L'écho
    duree_echo = signal_arrivee - signal_depart

    # Calculer La distance en fonction de La durée de L'écho
    distance = (duree_echo * 0.0343) / 2

    return distance

while True:
    # Mesurer La distance avec Le capteur HC-SR04
    distance = mesure_distance()

    # Afficher La distance mesurée
    print("Distance : {:.2f} cm".format(distance))

    # Attendre avant de refaire une mesure
    utime.sleep(1)
```

Explication du code

- Importez les modules nécessaires, notamment `machine` pour accéder aux fonctionnalités du Raspberry Pi Pico et `utime` pour la gestion du temps.
- Définissez les broches GPIO pour le déclenchement (Trig) et l'écho (Echo) du capteur HC-SR04.
- Définissez une fonction `mesure_distance()` pour effectuer la mesure de distance.
- Dans la fonction `mesure_distance()`, générez une impulsion de déclenchement en mettant la broche de déclenchement à l'état bas puis haut puis à nouveau bas.
- Attendez le signal d'écho en mesurant le temps auquel l'écho commence et se termine.

Calculez la durée de l'écho en soustrayant le temps de départ du temps d'arrivée. 7. Calculez la distance en utilisant la formule de conversion de la durée de l'écho en distance (en supposant une vitesse du son de 343 m/s).

- Dans la boucle principale, appelez la fonction `mesure_distance()` pour mesurer la distance avec le capteur HC-SR04.
- Affichez la distance mesurée et attendez avant de refaire une mesure.

Utilisation du capteur de distance HC-SR04 avec un module open source

► Installation du module open source

- Téléchargez le module open source `hc_sr04` pour MicroPython à partir du dépôt GitHub officiel (<https://github.com/rsc1975/micropython-hcsr04>).
- Copiez les fichiers `hc_sr04.py` et [pulseio.py](#) sur votre Raspberry Pi Pico.

Exemple de code pour la mesure de distance avec le capteur HC-SR04

```

import machine
import utime
from hc_sr04 import HCSR04

# Définir les broches GPIO pour le déclenchement et l'écho
trigger_pin = machine.Pin(2)
echo_pin = machine.Pin(3)

# Créer une instance du capteur HC-SR04
capteur_distance = HCSR04(trigger_pin, echo_pin)

while True:
    # Mesurer la distance avec le capteur HC-SR04
    distance = capteur_distance.distance_cm()

    # Afficher la distance mesurée
    print("Distance : {:.2f} cm".format(distance))

    # Attendre avant de refaire une mesure
    utime.sleep(1)

```

► Explication du code

- Importez les modules nécessaires, notamment machine pour accéder aux fonctionnalités du Raspberry Pi Pico et utime pour la gestion du temps.
- Importez la classe HCSR04 du module hc_sr04 pour utiliser le capteur HC-SR04.
- Définissez les broches GPIO pour le déclenchement (Trig) et l'écho (Echo) du capteur HC-SR04.
- Créez une instance du capteur HC-SR04 en passant les broches GPIO comme arguments.
- Dans la boucle principale, appelez la méthode distance_cm() de l'instance du capteur pour mesurer la distance.
- Affichez la distance mesurée et attendez avant de refaire une mesure.

Ressources supplémentaires

- Tutoriel vidéo sur l'utilisation du capteur de distance HC-SR04 avec MicroPython et Raspberry Pi Pico : <https://youtu.be/YAF3o3SsCmk>
- Dépôt GitHub du module open source hc_sr04 pour MicroPython : <https://github.com/rsc1975/micropython-hcsr04>

Ces instructions vous permettront de mesurer la distance à l'aide du capteur HC-SR04 en utilisant un module open source avec MicroPython sur votre Raspberry Pi Pico.

Ressources supplémentaires

- Tutoriel vidéo sur l'utilisation du capteur de distance HC-SR04 avec MicroPython et Raspberry Pi Pico : <https://youtu.be/YAF3o3SsCmk>

- Bibliothèque hcsr04 pour MicroPython sur GitHub : <https://github.com/rsc1975/micropython-hcsr04>

Ces instructions vous permettront de mesurer la distance à l'aide du capteur HC-SR04 en utilisant MicroPython sur votre Raspberry Pi Pico.

KY-004: Bouton

Cet article présente le KY-004, un bouton poussoir facile à utiliser.

Le module KY-004 est un bouton poussoir simple qui peut être utilisé dans de nombreux projets avec un Raspberry Pi Pico en MicroPython. Voici un exemple de code qui montre comment utiliser le KY-004 pour allumer et éteindre une LED en appuyant sur le bouton.

► Matériel nécessaire :

- Raspberry Pi Pico.
- Module KY-004 Button.
- Le feu rouge
- Câbles de connexion et breadboard.

► Connexion des composants :

- Connectez les LED des feux de circulation
- Connectez la broche **S** (Signal) du KY-004 à une autre broche GPIO (par exemple GPIO3).
- Connectez la broche **-** (GND) du KY-004 à GND de la Pico.

► Code MicroPython :

```
from machine import Pin
import utime

# Configuration de La broche pour La LED et Le bouton
led_rouge = Pin(2, Pin.OUT)
button = Pin(3, Pin.IN)

led_state = False

def toggle_led():
    global led_state
    led_state = not led_state
    led_rouge.value(led_state)

# Boucle principale
while True:
    if button.value() == 1: # Vérifie si Le bouton est appuyé
        toggle_led()       # Change L'état de La LED
        utime.sleep(0.5)    # Anti-rebond (debounce), évite Les changements d'état multipl
```

► Explication :

- La broche de la LED est configurée en sortie (`Pin.OUT`), et la broche du bouton en entrée (`Pin.IN`).
- La fonction `toggle_led` change l'état de la LED à chaque appel.
- Dans la boucle principale, le programme vérifie si le bouton est appuyé. Si c'est le cas, il appelle `toggle_led` pour changer l'état de la LED.
- Un délai de 0.5 seconde après chaque appui sur le bouton évite les rebonds (changement d'état multiple dû à la pression physique du bouton).

Ce code est un bon exemple pour apprendre les bases de la lecture des entrées numériques et le contrôle des sorties numériques en MicroPython.

Quel est le problème avec ce code?

Pour améliorer la gestion du bouton, essaie cette version: [bouton avec lecture unique](#)

Les micro-contrôleurs

*Un microcontrôleur est une unité de commande intégrée utilisée pour **contrôler le fonctionnement d'un robot**. Il s'agit d'un petit ordinateur sur une seule puce (SoC) qui combine un **microprocesseur**, de la **mémoire** et des **interfaces d'entrée/sortie**.*

En tant qu'unité de commande, le microcontrôleur **exécute les programmes et les instructions** qui définissent le comportement du robot. Il reçoit des informations provenant des capteurs du robot, traite ces données et envoie des commandes aux actionneurs pour effectuer des actions spécifiques.

Les microcontrôleurs sont souvent utilisés dans les robots en raison de leur **taille compacte**, de leur **faible consommation d'énergie** et de leur capacité à exécuter des tâches en **temps réel**. Ils offrent une solution intégrée pour contrôler les mouvements, les interactions avec l'environnement et les communications du robot.

Les microcontrôleurs peuvent être programmés à l'aide de différents langages, tels que **C/C++**, Python, **Micro-Python**, ou des langages de programmation visuels basés sur des blocs. Les programmeurs écrivent des codes qui spécifient le comportement du robot, y compris la prise de décision, les mouvements, la détection d'obstacles et les réponses aux événements.

En résumé, un microcontrôleur est l'**unité de commande centrale** d'un robot, responsable de l'exécution des programmes, de la gestion des capteurs et des actionneurs, et de la coordination des opérations pour permettre au robot d'accomplir des tâches spécifiques.

Micro:bit :

- Le micro:bit est un microcontrôleur largement utilisé pour l'apprentissage de la robotique, en particulier pour les débutants.
- Il est petit, abordable et **dispose de nombreux capteurs intégrés** tels que l'accéléromètre, le magnétomètre et les boutons.
- Le micro:bit utilise une programmation par blocs conviviale, ce qui le rend facile à apprendre pour les débutants.
- Il est souvent utilisé pour créer des projets robotiques simples tels que des robots suiveurs de ligne ou des robots éviteurs d'obstacles.

[En savoir plus \(notes de cours\)](#)

ESP8266 :

- L'ESP8266 est un microcontrôleur WiFi populaire utilisé dans la robotique et l'Internet des objets (IoT).
- Il dispose d'une connectivité WiFi intégrée, ce qui permet aux robots de se connecter à des réseaux sans fil et d'interagir avec d'autres appareils.
- L'ESP8266 est compatible avec plusieurs langages de programmation, tels que Arduino IDE et MicroPython.
- Il est souvent utilisé pour des projets de robotique avancés nécessitant une communication sans fil, tels que des robots contrôlés par smartphone ou des robots de surveillance.

Raspberry Pi Pico :

- Le Raspberry Pi Pico est un microcontrôleur développé par la Fondation Raspberry Pi.
- Il est basé sur le microcontrôleur RP2040 et dispose de nombreuses fonctionnalités, telles que des broches d'E/S polyvalentes et une programmation en MicroPython et C/C++.
- Le Raspberry Pi Pico est abordable, polyvalent et peut être utilisé pour des projets robotiques de taille moyenne à avancée.
- Il est souvent utilisé pour créer des robots autonomes, des robots avec des capteurs avancés et des projets de robotique embarquée.

[En savoir plus \(notes de cours\)](#)

LEGO Mindstorms :

- LEGO Mindstorms est une **plateforme de robotique éducative** très populaire pour les enfants et les débutants.
- Elle utilise des briques LEGO spéciales, des moteurs et des capteurs pour construire des robots.
- Les robots LEGO Mindstorms peuvent être programmés à l'aide d'un logiciel convivial basé sur des blocs graphiques.
- Cette plateforme est idéale pour enseigner les bases de la robotique et de la programmation, en mettant l'accent sur la créativité et la construction physique.

Thymio :

- Thymio est un **robot éducatif** conçu pour l'apprentissage de la robotique et de la programmation.
- Il est doté de capteurs et d'actionneurs, tels que des capteurs de proximité, des détecteurs de lumière et des moteurs.
- Thymio peut être programmé à l'aide d'un logiciel visuel ou d'un langage de programmation textuel, ce qui permet d'explorer différents niveaux de complexité.
- Ce robot est largement utilisé dans les écoles pour enseigner la programmation, la logique et les concepts de base de la robotique.

Ces microcontrôleurs offrent une excellente opportunité d'apprentissage pour les étudiants dans le domaine de la robotique. Ils sont abordables, faciles à programmer et offrent des fonctionnalités adaptées à différents niveaux de compétence.

Le Raspberry Pi Pico

Sur cette page, nous allons explorer les différentes broches du Pico, leur numéro, leur fonction (analogique, PWM, numérique), et comment les utiliser en MicroPython.

Broches GPIO du Raspberry Pi Pico

Raspberry-Pico
microcontrôleur RP2040 à 40 broches

USB Type B
- Alimentation
- Programmation
- Disque Flash
- USB1.1 Host & Client

VBUS alimenté (connecté sur USB)

MicroPython REPL

GP24
GP25
GP0
GP1
GP2
GP3
GP4
GP5
GP6
GP7
GP8
GP9
GP10
GP11
GP12
GP13
GP14
GP15
GP16
GP17
GP18
GP19
GP20
GP21
GP22
GP23
GP28
GP27
GP26
GP25
GP24
GP23
GP22
GP21
GP20
GP19
GP18
GP17
GP16
GP15
GP14
GP13
GP12
GP11
GP10
GP9
GP8
GP7
GP6
GP5
GP4
GP3
GP2
GP1
GP0

PWM A[0]
PWM B[0]
PWM A[1]
PWM B[1]
PWM A[2]
PWM B[2]
PWM A[3]
PWM B[3]
PWM A[4]
PWM B[4]
PWM A[5]
PWM B[5]
PWM A[6]
PWM B[6]
PWM A[7]
PWM B[7]

UART TX
UART RX

LED

USB

BOOTSEL

3V3_EN

3.3V 300mA max

ADC_VREF = 3.3V filtré!

ADC2
AGND = GND
ADC1
ADC0

Entrées Analog.
3.3V Max!

PWM A[6]
PWM B[5]
PWM A[5]

LRCLK_PWM0
BCK_PWM1
DIN

125

SD_DAT3/CD
SD_DAT2
SD_DAT1
SD_DAT0
SD_CMD

SCL I2C(1)
SDA I2C(1)

SCL I2C(0)
SDA I2C(0)

SWCLK
SWDIO

Position alternative pour I2C

Voici une des nombreuses positions alternatives pour I2C(0) et I2C(1).
Un tel bus peut être créé en ajoutant les broches sda et scl souhaitées.

```
i2c_0 = I2C( 0, sda=Pin(16), scl=Pin(17) )
i2c_1 = I2C( 1, sda=Pin(18), scl=Pin(19) )
```

RP2040 - Double Coeur
- ARM Cortex M0+ (FPU) @ 133 Mhz
- 264 Kio SRAM
- 2Mo Flash externe (QSI)
- 128 KB Système de fichier MicroPython (En flash interne)

shop.mchobby.be

v0.2

Utilisation des Broches en MicroPython

► Configuration d'une broche en mode numérique

Pour configurer une broche en mode numérique (entrée ou sortie), vous pouvez utiliser la bibliothèque `machine` en MicroPython. Par exemple, pour configurer la broche GP0 en tant que sortie, vous pouvez utiliser le code suivant :

```
import machine

# Configurer GP0 en sortie
broche_gp0 = machine.Pin(0, machine.Pin.OUT)
```

MicroPython offre plusieurs méthodes pour contrôler la tension de sortie :

```
from machine import Pin

Pin.on() et Pin.off()
Pin.high() et Pin.low()
Pin.value(valeur)
```

Les noms de fonctions sont explicites :

```
pin_led = Pin(16, mode=Pin.OUT)

# Imposer une tension de 3.3V en sortie (état logique haut)
pin_led.on()
pin_led.high()
pin_led.value(1)

# Imposer une tension de 0V en sortie (état logique bas)
pin_led.off()
pin_led.low()
pin_led.value(0)
```

► Utilisation des broches en mode PWM

Le Raspberry Pi Pico prend en charge la modulation de largeur d'impulsion (PWM) sur certaines broches. Cela permet de contrôler la vitesse des moteurs, la luminosité des LED, etc. Voici comment configurer une broche en mode PWM :

```
import machine

# Configurer GP2 en mode PWM
broche_gp2 = machine.Pin(2)
pwm = machine.PWM(broche_gp2)
```

► Lecture de valeurs analogiques

Pour lire des valeurs analogiques à partir des broches GPIO, vous pouvez utiliser l'ADC (Convertisseur Analogique-Numérique) intégré au Pico.

Note/Rappel: Le RPi Pico possède 4 broches analogiques: GP26->29

Voici comment configurer une broche pour lire une valeur analogique :

```
import machine

# Configurer GP26 pour la lecture analogique
broche_gp26 = machine.Pin(26)
adc = machine.ADC(broche_gp26)

# Lire la valeur analogique
valeur_analogique = adc.read_u16()
```

Le code suivant permet de lire la valeur d'un potentiomètre analogique:

```
from machine import ADC, Pin
from utime import sleep
pot = ADC(26)
conversion_factor = 3.3 / 65535

while True:
    raw = pot.read_u16()
    volts = raw * conversion_factor
    print('Raw: {} '.format(raw), 'Voltage {:.1f}V'.format(volts))
    sleep(1)
```

Et ce code permet de changer la fréquence de clignotement d'une LED en fonction de la valeur lue sur un potentiomètre:

```

from machine import ADC, Pin
from utime import sleep

pot = ADC(26)
led = Pin(25, Pin.OUT)

def map(s, a1, a2, b1, b2):
    return b1 + (s - a1) * (b2 - b1) / (a2 - a1)

while True:
    raw = pot.read_u16()
    delay = map(raw, 0, 65535, 0, 1)
    print('Raw: {} '.format(raw), 'Delay {:.1f}s'.format(delay))
    led.value(1)
    sleep(delay)
    led.value(0)
    sleep(delay)

```

Conclusion

Les broches GPIO du Raspberry Pi Pico offrent de nombreuses possibilités pour interagir avec le monde physique en utilisant MicroPython. Que ce soit pour contrôler des LED, des moteurs, ou pour lire des capteurs analogiques, vous avez maintenant les connaissances de base pour commencer à créer vos propres projets avec le Pico. Explorez davantage et expérimentez avec les différentes broches pour donner vie à vos idées.

MicroPython sur RPi Pico avec Thonny Python

///

Prérequis

- Assurez-vous d'avoir un Raspberry Pi Pico fonctionnel et un câble USB pour le connecter à votre ordinateur.
- Téléchargez et installez l'IDE Thonny Python sur votre ordinateur à partir du site officiel de Thonny (<https://thonny.org/>).

Installation du firmware MicroPython

- Visitez le site officiel de MicroPython pour Raspberry Pi Pico (<https://micropython.org/download/rp2-pico/>) et téléchargez le dernier firmware MicroPython (.uf2) pour le Raspberry Pi Pico.

Connexion du Raspberry Pi Pico

- Connectez votre Raspberry Pi Pico à votre ordinateur à l'aide du câble USB. Assurez-vous que le commutateur "BOOTSEL" est enfoncé pour mettre le Pico en mode Bootloader.

Flashage du firmware MicroPython

- Ouvrez l'IDE Thonny Python sur votre ordinateur.
- Dans Thonny, cliquez sur le bouton "Sélectionner l'interpréteur" en haut à droite.
- Dans la fenêtre de sélection de l'interpréteur, choisissez "Raspberry Pi Pico (MicroPython)".
- Maintenant, dans Thonny, cliquez sur le bouton "Flasher" en haut.
- Dans la fenêtre de flashage, sélectionnez le firmware MicroPython (.uf2) que vous avez précédemment téléchargé.
- Cliquez sur "OK" pour flasher le firmware sur votre Raspberry Pi Pico.

Écriture et exécution du code MicroPython

- Une fois le firmware MicroPython installé, le Raspberry Pi Pico redémarrera automatiquement.
- Dans Thonny, vous verrez le Pico répertorié comme un périphérique connecté.
- Créez un nouveau fichier en cliquant sur "Nouveau" dans le menu Fichier.
- Écrivez votre code MicroPython dans le nouvel onglet du fichier.
- Pour exécuter le code, cliquez sur le bouton "Exécuter" ou appuyez sur F5.

Interaction avec le Raspberry Pi Pico

- Vous pouvez utiliser la console REPL (Read-Eval-Print Loop) pour interagir avec le Raspberry Pi Pico en temps réel.
- Dans Thonny, cliquez sur l'icône de console en bas de la fenêtre pour ouvrir la console REPL.
- Vous pouvez saisir des commandes directement dans la console pour interagir avec les broches GPIO, les capteurs ou les actionneurs connectés au Raspberry Pi Pico.

Débogage et surveillance

- Thonny Python offre également des fonctionnalités de débogage et de surveillance pour vous aider à identifier et corriger les erreurs dans votre code MicroPython.
- Vous pouvez placer des points d'arrêt, examiner les variables et suivre l'exécution pas à pas pour comprendre le comportement de votre code.

Ces étapes vous permettront de configurer et d'utiliser MicroPython sur votre Raspberry Pi Pico en utilisant l

Raspberry Pi Pico et les Ports GPIO

///

- Le Raspberry Pi Pico est un microcontrôleur à faible coût développé par la Fondation Raspberry Pi.
- Il est basé sur le microcontrôleur RP2040 et offre de nombreuses fonctionnalités pour les projets de robotique et d'électronique.



Les Ports GPIO (General-Purpose Input/Output)

- Le Raspberry Pi Pico dispose de **26 broches GPIO (0 à 25)** qui peuvent être configurées en tant qu'**entrées ou sorties** pour interagir avec d'autres composants électroniques.
- Les broches GPIO peuvent être utilisées pour **lire des signaux provenant de capteurs externes ou pour contrôler des actionneurs** tels que des moteurs ou des LED.
- Chaque broche GPIO peut être configurée individuellement et prend en charge différents modes de fonctionnement, tels que l'**entrée**, la **sortie**, la **PWM** (modulation de largeur d'impulsion), l'I2C, le SPI, etc.

Utilisation des Ports GPIO

- Configuration des broches : Avant d'utiliser une broche GPIO, il est nécessaire de la **configurer dans le mode de fonctionnement souhaité** en utilisant des bibliothèques de programmation appropriées, telles que RP2PIO ou RP2GPIO pour le Raspberry Pi Pico.
- Lecture des broches d'entrée : Les broches GPIO configurées en tant qu'entrées peuvent être utilisées pour lire des signaux provenant de capteurs externes. On peut les lire en utilisant des fonctions de lecture appropriées dans le langage de programmation utilisé, telles que `input()` en Python.
- Contrôle des broches de sortie : Les broches GPIO configurées en tant que sorties peuvent être utilisées pour contrôler des actionneurs, tels que des moteurs ou des LED. On peut les contrôler en utilisant des fonctions d'écriture appropriées dans le langage de programmation utilisé, telles que `output()` en Python.

Exemples d'utilisation des Ports GPIO

- Contrôle de LED : On peut connecter une LED à une broche GPIO configurée en tant que sortie et utiliser la fonction d'écriture pour allumer ou éteindre la LED.
- Lecture de bouton : On peut connecter un bouton à une broche GPIO configurée en tant qu'entrée et utiliser la fonction de lecture pour détecter si le bouton est enfoncé ou relâché.
- Contrôle de moteur : On peut utiliser les broches GPIO configurées en tant que sorties pour contrôler la vitesse et la direction d'un moteur en utilisant la modulation de largeur d'impulsion (PWM) pour créer des signaux analogiques simulés.

Modes Analogique, Digital et PWM

Mode Analogique

- Le mode analogique est utilisé pour représenter et manipuler des **signaux continus** qui peuvent prendre une gamme de valeurs différentes.
- Dans le contexte des microcontrôleurs, le mode analogique permet de **mesurer ou de générer des signaux analogiques**.
- Lorsqu'un microcontrôleur est configuré en mode analogique, il peut **lire des tensions** provenant de capteurs analogiques tels que des capteurs de température ou de lumière.
- Les microcontrôleurs peuvent également générer des signaux analogiques en utilisant des *convertisseurs numérique-analogique* (CNA) pour contrôler des dispositifs tels que des amplificateurs ou des moteurs.

Mode Digital

- Le mode digital est utilisé pour représenter des signaux qui ne peuvent prendre que **deux valeurs distinctes : 0 et 1 (ou HIGH et LOW)**.
- Les microcontrôleurs sont capables de traiter des signaux digitaux, ce qui leur permet de communiquer avec d'autres composants électroniques tels que des capteurs, des actionneurs ou d'autres microcontrôleurs.
- Les broches GPIO configurées en tant que sorties digitales peuvent envoyer des signaux à d'autres composants, tandis que les broches configurées en tant qu'entrées digitales peuvent recevoir des signaux provenant d'autres sources.

Modulation de Largeur d'Impulsion (PWM)

- La modulation de largeur d'impulsion (PWM) est une technique utilisée pour **simuler des signaux analogiques en utilisant des signaux digitaux**.
- Elle est couramment utilisée pour contrôler la vitesse ou l'intensité de dispositifs tels que des moteurs, des LED ou des servomoteurs.
- Dans un signal PWM, **la largeur de l'impulsion varie dans le temps**, tandis que la période de l'impulsion reste constante.

- En ajustant le rapport cyclique (rapport entre la durée de l'impulsion et la période), on peut contrôler la moyenne pondérée du signal, ce qui permet de varier la vitesse ou l'intensité du dispositif contrôlé.

Utilisation dans la Robotique

- En robotique, les **signaux analogiques** sont souvent utilisés **pour mesurer des grandeurs** telles que la **température**, la **luminosité** ou la **distance**.
- Les signaux digitaux sont utilisés pour la communication entre composants électroniques et la **lecture d'états** de capteurs, tels que les **boutons** ou les **interrupteurs**.
- La modulation de largeur d'impulsion (**PWM**) est utilisée pour **contrôler précisément la vitesse des moteurs** ou la **luminosité des LED**, ce qui est essentiel dans de nombreux projets robotiques.

Il est important de comprendre ces différents modes et techniques, car ils permettent aux microcontrôleurs de s'adapter à différentes situations et de contrôler divers dispositifs dans les projets robotiques.

Ressources supplémentaires

- Documentation officielle du Raspberry Pi Pico : <https://www.raspberrypi.org/documentation/pico/>
- Tutoriels et exemples de projets : Le site officiel du Raspberry Pi Pico propose une variété de tutoriels et de projets pour vous aider à démarrer avec les ports GPIO et d'autres fonctionnalités du microcontrôleur.

Utilisation du Raspberry Pi Pico avec MicroPython

Dans cet article de cours, nous allons explorer comment utiliser le Raspberry Pi Pico avec MicroPython et vous montrer des exemples de programmes pour lire et écrire des valeurs sur les broches GPIO (General-Purpose Input/Output).

Le Raspberry Pi Pico est un microcontrôleur puissant et abordable qui peut être programmé avec MicroPython, un langage de programmation convivial pour les débutants.

Prérequis

Avant de commencer, assurez-vous d'avoir les éléments suivants à votre disposition :

- Un Raspberry Pi Pico
- Un ordinateur avec MicroPython installé (vous pouvez télécharger MicroPython sur le site officiel)
- Un câble micro USB pour la connexion entre l'ordinateur et le Raspberry Pi Pico

Installation de MicroPython

Ceci est une procédure d'installation manuelle que nous n'avons pas vue en classe. Je la laisse pour que vous sachiez que ça existe, mais il ne faut pas la connaître pour le cours.

1. Téléchargez la dernière version de MicroPython depuis le site officiel (<https://micropython.org/>).
2. Connectez votre Raspberry Pi Pico à votre ordinateur à l'aide du câble micro USB. Le Pico devrait être reconnu comme un périphérique de stockage USB.
3. Copiez le fichier MicroPython que vous avez téléchargé sur le Pico. Il doit avoir une extension ".uf2".
4. Déconnectez le Pico de votre ordinateur, puis reconnectez-le. Vous verrez un nouveau périphérique de stockage appelé "RPI-RP2".
5. Supprimez le fichier "CURRENT.UF2" s'il existe et renommez votre fichier MicroPython téléchargé en "CURRENT.UF2". Cela permettra au Pico de démarrer automatiquement avec MicroPython.
6. Déconnectez à nouveau le Pico de l'ordinateur, puis reconnectez-le. Il devrait maintenant exécuter MicroPython.

Programme de base

Commençons par un exemple simple de programme en MicroPython pour allumer et éteindre une LED connectée à une broche GPIO du Raspberry Pi Pico.

```
import machine
import utime

# Définir la broche GPIO à utiliser
led = machine.Pin(25, machine.Pin.OUT)

# Boucle infinie pour clignoter la LED
while True:
    led.toggle() # Inverse l'état de la LED
    utime.sleep(1) # Attendre 1 seconde
```

Dans ce programme, nous utilisons le module `machine` pour gérer les broches GPIO. Nous définissons la broche 25 comme une sortie (`machine.Pin.OUT`). Ensuite, nous utilisons une boucle infinie pour alterner l'état de la LED entre allumé et éteint toutes les secondes à l'aide de la méthode `toggle()` .

Lecture de valeurs analogiques

Vous pouvez également lire des valeurs analogiques à partir de broches GPIO pour mesurer des grandeurs telles que la luminosité avec un capteur de lumière. Voici un exemple pour lire une valeur analogique à partir de la broche 26 :

```
import machine

# Définir la broche GPIO à utiliser
analog_pin = machine.Pin(26)

# Configurer l'ADC (Convertisseur Analogique-Numérique)
adc = machine.ADC(analog_pin)

# Lire et imprimer la valeur analogique
while True:
    valeur = adc.read_u16() # Lecture de la valeur analogique (0-65535)
    print("Valeur analogique : ", valeur)
```

Dans cet exemple, nous utilisons le module `machine` pour configurer l'ADC (Convertisseur Analogique-Numérique) sur la broche 26. La valeur analogique est lue à l'aide de la méthode `read_u16()` et affichée dans la console.

Conclusion

Le Raspberry Pi Pico avec MicroPython offre une excellente plateforme pour l'apprentissage de la programmation et de l'électronique. Vous pouvez créer une variété de projets intéressants en utilisant les broches GPIO pour contrôler des composants électroniques. N'hésitez pas à explorer davantage et à créer vos propres projets en utilisant ces concepts de base.

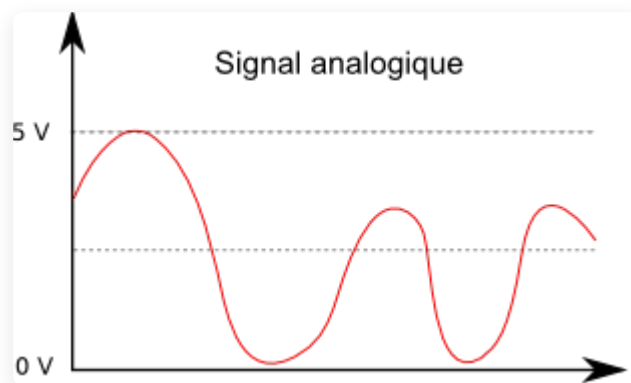
Les différents types de signaux

Les microcontrôleurs sont des composants essentiels dans de nombreux systèmes électroniques, des appareils ménagers aux robots industriels. Pour interagir avec leur environnement, ces microcontrôleurs utilisent différents types de signaux. Dans cet article, nous allons explorer en détail les trois principaux types de signaux utilisés : les signaux analogiques, les signaux digitaux et les signaux à modulation de largeur d'impulsion (PWM). Nous verrons comment ils fonctionnent, leurs applications et les bonnes pratiques pour les utiliser.

1. Signaux Analogiques

► Qu'est-ce qu'un Signal Analogique ?

Un signal analogique est un **signal continu** qui peut prendre une **infinité de valeurs** dans une plage donnée. Contrairement aux signaux digitaux, qui ne peuvent prendre que des valeurs discrètes (0 ou 1), les signaux analogiques peuvent représenter des variations continues, comme la température, la luminosité ou le son.



► Exemple de Signal Analogique

Imaginez un capteur de température analogique. La sortie de ce capteur est une tension continue qui varie en fonction de la température mesurée. Par exemple, à 20°C, la sortie pourrait être de 2V, à 25°C, elle pourrait être de 2.5V, et ainsi de suite.

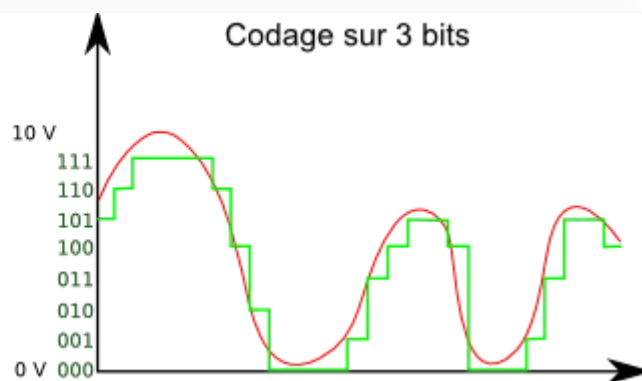
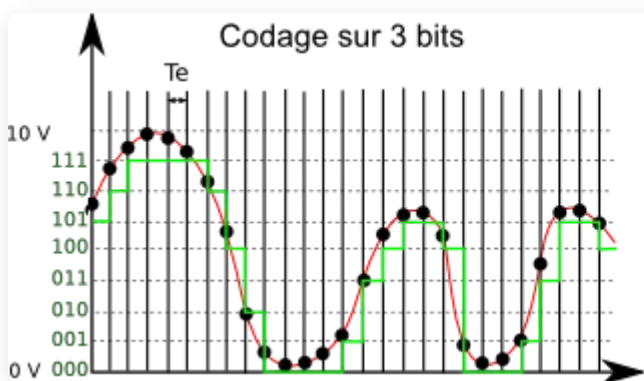
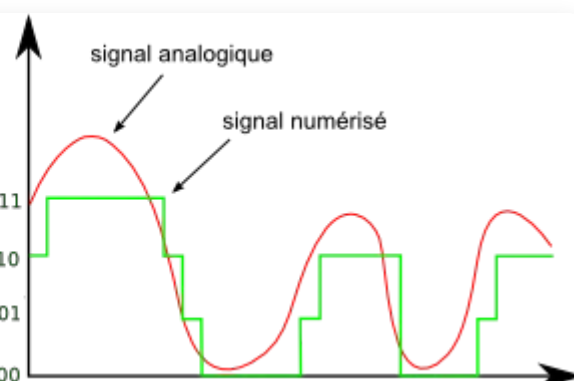
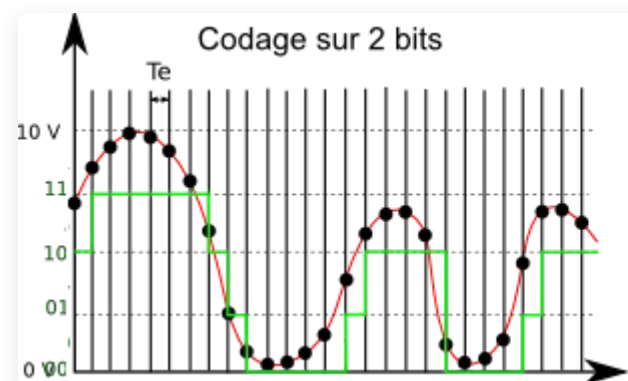
► Utilisation dans les Microcontrôleurs

Les microcontrôleurs ne peuvent pas traiter directement les signaux analogiques. Ils doivent **convertir ces signaux** en valeurs numériques à l'aide d'un **convertisseur analogique-numérique (ADC)**. Une fois le signal converti, le microcontrôleur peut le traiter de manière numérique.

L'ADC **convertit une tension analogique continue** en une valeur numérique discrète que le microcontrôleur peut traiter. Cette conversion se fait en **échantillonnant** le signal analogique à des intervalles de temps réguliers et en attribuant à chaque échantillon une valeur numérique correspondante.

Chaque échantillon est attribué à la valeur numérique la plus proche parmi un ensemble de valeurs discrètes possibles.

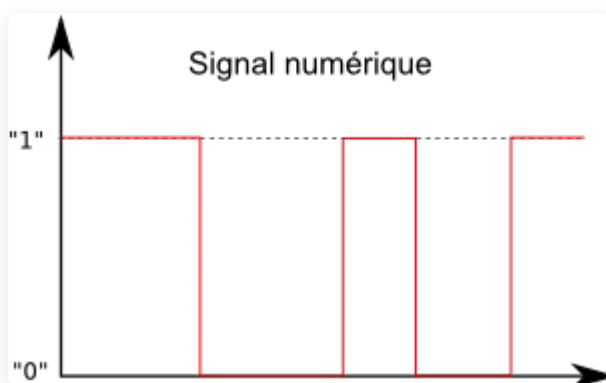
Une caractéristique important des ADC est leur **résolution** : C'est le **nombre de bits de la sortie numérique**. Une résolution de **10 bits** signifie que la sortie peut prendre **1024** valeurs différentes (2^{10}). **Plus la résolution est élevée, plus la conversion est précise.**



2. Signaux Digitaux

► Qu'est-ce qu'un Signal Digital ?

Un signal **digital**, ou **numérique**, ne peut prendre que deux valeurs distinctes : **0 (faible)** ou **1 (élevé)** (en tout cas en ce qui concerne les i/o des micro-contrôleurs). Ces signaux sont utilisés pour représenter des états binaires ou des données discrètes.



► Exemple de Signal Digital

Considérons un **interrupteur** connecté à un microcontrôleur. Lorsque l'interrupteur est fermé, la sortie est à 1 (élevé), et lorsqu'il est ouvert, la sortie est à 0 (faible).

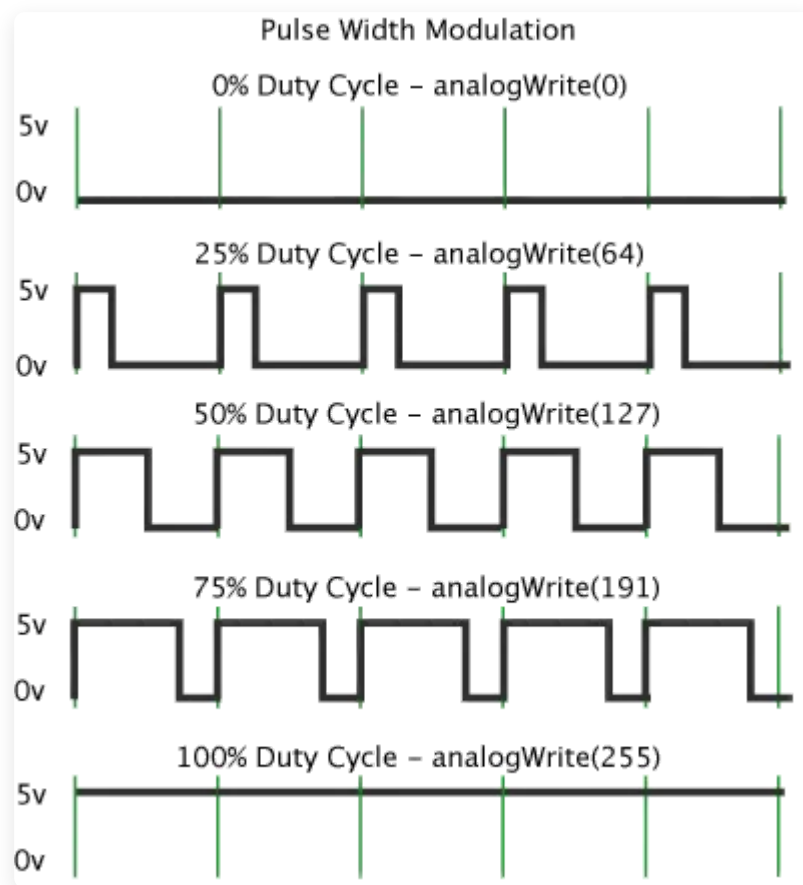
► Utilisation dans les Microcontrôleurs

Les microcontrôleurs traitent naturellement les signaux digitaux. Ils peuvent lire les états des broches d'entrée/sortie (I/O) et exécuter des actions en fonction de ces états.

3. Signaux à Modulation de Largeur d'Impulsion (PWM)

► Qu'est-ce que le PWM ?

La **modulation de largeur d'impulsion** (GB *PWM* ou *pulse width modulation*) est une technique où la largeur des impulsions d'un signal est modulée pour représenter une **valeur analogique moyenne**. Le **rapport cyclique** (GB *duty cycle*) détermine la **proportion de temps** pendant laquelle le signal est à l'état haut (1) par rapport à l'état bas (0).



Le graphe ci-dessus montre l'équivalence entre différents duty cycles et la valeur équivalente en analogique (analogWrite).

► Exemple de PWM

Supposons que nous ayons une LED contrôlée par un signal PWM. En modifiant le rapport cyclique, nous pouvons contrôler la luminosité de la LED. Un rapport cyclique de 50% signifie que le signal est haut pendant 50% du temps et bas pendant les 50% restants, ce qui correspond à une luminosité moyenne de la LED.

► Utilisation dans les Microcontrôleurs

Les microcontrôleurs peuvent générer des signaux PWM à l'aide de modules dédiés. Ces signaux sont souvent utilisés pour le contrôle de la puissance, comme dans les moteurs, les LEDs ou les régulateurs de tension.

Conclusion

Les microcontrôleurs utilisent des signaux **analogiques**, **digitaux** et **PWM** pour interagir avec leur environnement. Chaque type de signal a ses propres caractéristiques, avantages et inconvénients. Les signaux analogiques permettent de représenter des valeurs continues, les signaux digitaux sont parfaits pour les états binaires et la communication, et les signaux PWM offrent une solution efficace pour le contrôle de puissance et la génération de valeurs analogiques moyennes.

En comprenant comment utiliser ces différents types de signaux, vous pouvez concevoir des systèmes plus robustes et efficaces.

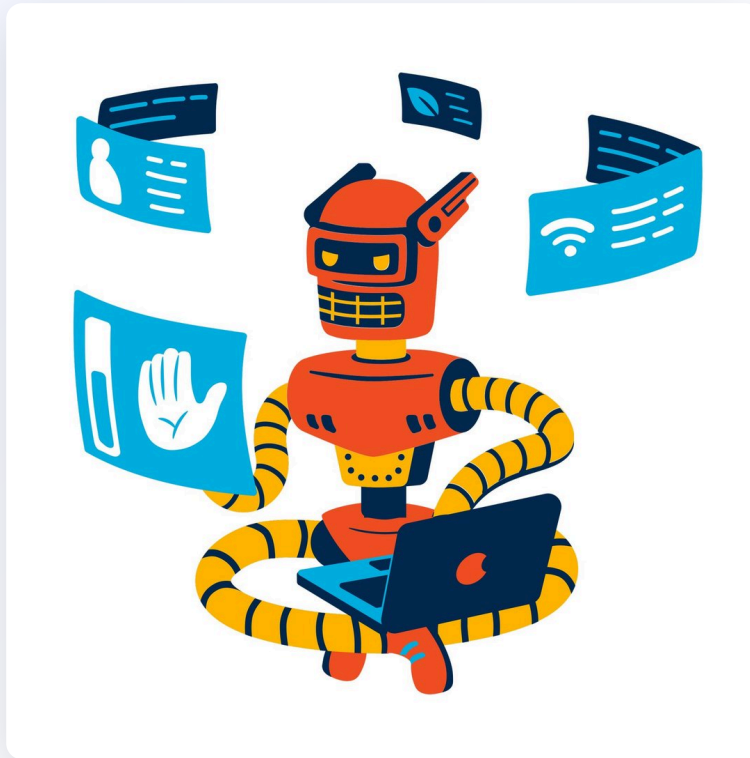
CANSAT

Liens utiles

Liens utiles

- <https://www.innoviris.brussels/fr/program/cansat-belgium>
- Wiki Cansat: <https://wiki.mchobby.be/index.php?title=ENG-CANSAT-PICO-MISSION1-RECEIVE>
- <https://meteologix.com/be/observations/pressure-qnh.html>
- Altitude Jodoigne: <https://fr-be.topographic-map.com/map-vncz/Jodoigne/?center=50.72818%2C4.87376&zoom=16&popup=50.72866%2C4.87617>

Documentations



GESTION DE PROJETS

Introduction à la Gestion de Projet

La **gestion de projet** est l'ensemble des méthodes, outils et pratiques permettant de **planifier, organiser et piloter un projet** afin d'atteindre des objectifs précis dans un délai et avec des ressources définies. Un projet se distingue par son caractère **temporaire** (il a un début et une fin) et par sa **finalité unique** (il vise à produire un livrable spécifique, comme un logiciel, une application ou un service).

En informatique, la gestion de projet est essentielle pour **structurer le développement de logiciels, la mise en place d'infrastructures ou la conception de systèmes complexes**. Sans une gestion efficace, les projets risquent d'être mal organisés, de dépasser les délais ou le budget, ou encore de ne pas répondre aux attentes des utilisateurs. Que ce soit avec des approches **classiques** (en cascade) ou **agiles**, bien gérer un projet informatique permet d'optimiser le travail des équipes, d'anticiper les risques et d'assurer la réussite du produit final. 🚀

Méthodes agiles

L'Agilité est une approche moderne de gestion de projet qui repose sur la flexibilité, l'adaptabilité et la collaboration. Contrairement aux méthodes traditionnelles, qui suivent un plan rigide du début à la fin, les méthodes Agiles permettent d'ajuster le projet en cours de route en fonction des retours et des priorités changeantes.

Qu'est-ce que l'Agilité en gestion de projet ?

L'Agilité est une approche moderne de gestion de projet qui repose sur la flexibilité, l'adaptabilité et la collaboration. Contrairement aux méthodes traditionnelles, qui suivent un plan rigide du début à la fin, les méthodes Agiles permettent d'ajuster le projet en cours de route en fonction des retours et des priorités changeantes.

L'Agilité est particulièrement utilisée dans le développement logiciel, mais elle s'étend aujourd'hui à d'autres domaines comme le marketing, la gestion de produit ou encore l'industrie.

Les principes des méthodes Agiles

Les méthodes Agiles sont basées sur le **Manifeste Agile** (2001), qui repose sur **quatre valeurs fondamentales** :

1. **Les individus et leurs interactions** plus que les processus et les outils.
2. **Un logiciel fonctionnel** plus qu'une documentation exhaustive.
3. **La collaboration avec les clients** plus que la négociation contractuelle.
4. **L'adaptation au changement** plus que le suivi d'un plan rigide.

L'Agilité encourage donc une approche plus dynamique et humaine du travail, axée sur la satisfaction du client et l'amélioration continue.

Les principales méthodes Agiles

Plusieurs méthodologies appliquent les principes de l'Agilité. Voici les plus connues :

► 1 Scrum

- Fonctionne en **sprints** (cycles courts de 1 à 4 semaines).
- Rôles clés : **Scrum Master** (facilitateur), **Product Owner** (vision du produit) et **équipe de développement**.
- Utilise un **tableau Scrum** pour suivre les tâches et une **réunion quotidienne** (Daily Scrum) pour ajuster le travail.

► 2 Kanban

- Utilise un **tableau Kanban** pour visualiser l'avancement des tâches.

- Favorise un **flux de travail continu**, en limitant le nombre de tâches en cours (**WIP – Work In Progress**).
- Permet une grande flexibilité, car il n'impose pas de cycles rigides.

▶ **3 Extreme Programming (XP)**

- Axé sur les **bonnes pratiques du développement logiciel** (tests automatisés, programmation en binôme, intégration continue).
- Vise à améliorer la qualité du code et la réactivité aux changements.

▶ **4 Lean**

- Inspiré du **système de production de Toyota**.
- Objectif : **réduire les gaspillages** et optimiser la valeur apportée au client.
- Met l'accent sur l'amélioration continue et la simplification des processus.

Pourquoi utiliser une méthode Agile ?

- ✓ **Plus de flexibilité** : on peut adapter le projet aux évolutions et aux imprévus.
- ✓ **Une meilleure collaboration** : les équipes travaillent ensemble et avec les clients.
- ✓ **Livraison plus rapide** : des petites versions du projet sont livrées régulièrement.
- ✓ **Amélioration continue** : les erreurs sont corrigées rapidement et les processus sont optimisés au fur et à mesure.

Les méthodes Agiles permettent donc de **répondre plus efficacement aux besoins des clients** tout en favorisant un travail d'équipe plus fluide et motivant. 🚀

La méthode Kanban



Qu'est-ce que la méthode Kanban ?

La méthode **Kanban** est une approche de gestion du travail qui permet de visualiser, organiser et optimiser les flux de travail. Elle est particulièrement utilisée dans le domaine du développement logiciel, de la gestion de projet et même dans la gestion de tâches personnelles.

Kanban repose sur l'utilisation d'un **tableau Kanban**, qui est un outil visuel permettant de suivre l'avancement des tâches. L'objectif est d'améliorer l'efficacité et la fluidité du travail en identifiant rapidement les blocages et en limitant le travail en cours (**WIP – Work In Progress**).

Le principe de la méthode Kanban

Kanban repose sur quelques principes clés :

1. **Visualisation du travail** : chaque tâche est représentée sous forme de carte (ou post-it) placée sur un tableau, qui est divisé en plusieurs colonnes représentant les différentes étapes du processus.
2. **Limiter le travail en cours (WIP – Work In Progress)** : il ne faut pas surcharger une étape avec trop de tâches pour éviter les goulots d'étranglement.
3. **Gérer le flux** : l'objectif est de maintenir un flux de travail fluide et d'éviter les blocages.
4. **Rendre explicites les règles du processus** : chaque équipe doit définir clairement ses règles pour garantir une bonne organisation.

5. **Amélioration continue** : Kanban est une méthode évolutive, qui s'adapte aux besoins en identifiant et en corrigeant les inefficacités.

L'objectif de la méthode Kanban

L'objectif principal de Kanban est **d'améliorer la productivité et l'efficacité en rendant le travail plus fluide et transparent**. En utilisant Kanban, une équipe peut :

- Réduire le temps nécessaire pour accomplir une tâche.
- Identifier rapidement les blocages.
- Adapter le processus de travail aux besoins réels.
- Travailler de manière plus collaborative et efficace.

Comment mettre en œuvre Kanban de façon simple ?

► 1. Créer un tableau Kanban

Le tableau Kanban peut être **physique** (un tableau blanc avec des post-it) ou **numérique** (via des outils comme Trello, Jira, ou Notion). Il est divisé en colonnes qui représentent les différentes étapes du processus de travail.

Un exemple classique de tableau Kanban :

À FAIRE (TO DO)	EN COURS (IN PROGRESS)	TERMINÉ (DONE)
Rédiger le rapport	Corriger un bug	Publier l'article
Créer une maquette	Développer une fonctionnalité	Valider le projet

► 2. Ajouter les tâches sous forme de cartes

Chaque tâche est représentée sous forme d'une **carte** (ou post-it). Une carte contient généralement :

- Un titre clair de la tâche.
- Une description si nécessaire.
- Une personne assignée.
- Une date limite (optionnelle).

► 3. Déplacer les tâches au fur et à mesure de leur avancement

Les tâches avancent progressivement à travers les colonnes du tableau en fonction de leur état d'avancement. Par exemple :

1. Une nouvelle tâche commence dans la colonne "**À faire**".
2. Lorsqu'une personne commence à travailler dessus, elle la déplace dans "**En cours**".

3. Une fois la tâche terminée, elle est placée dans "**Terminé**".

► 4. Limiter le travail en cours (WIP)

Une règle importante de Kanban est de **limiter le nombre de tâches en cours**. Par exemple, si une équipe fixe une limite de 3 tâches en cours par personne, cela évite la surcharge et permet de mieux se concentrer sur les priorités.

► 5. Améliorer continuellement le processus

Kanban encourage une réflexion continue sur l'amélioration du flux de travail. Quelques questions à se poser régulièrement :

- Y a-t-il des blocages fréquents dans une étape ?
- Une colonne est-elle souvent surchargée ?
- Pouvons-nous mieux répartir les tâches ?
- Le tableau reflète-t-il bien notre manière de travailler ?

En ajustant les règles et les limites, l'équipe optimise progressivement son travail.

Conclusion : ce qu'il faut retenir

- ✓ **Kanban est une méthode visuelle de gestion du travail** qui repose sur un tableau avec des tâches organisées en colonnes.
- ✓ Son objectif est **d'optimiser le flux de travail et d'améliorer l'efficacité** en limitant le travail en cours.
- ✓ Il suffit de **créer un tableau**, d'y ajouter des tâches sous forme de cartes et de les faire avancer d'une colonne à l'autre.
- ✓ La méthode encourage une **amélioration continue** en adaptant les règles aux besoins de l'équipe.

Kanban est une méthode simple mais puissante, idéale pour mieux organiser ton travail et celui de ton équipe ! 🚀