Python

Sommaire

- 2.5. Le Package Manager

3. Les bases

- 3.1. Les notations / particularitées syntaxique
  - 3.2. Les variables et ses types
  - 3.3. Les fonctions de bases
    - 3.3.1. Gérer la console
    - 3.3.2. Les opérations et calculs
  - 3.4. Les commentaires
    - 3.4.1. Les commentaires en fin de ligne
    - 3.4.2. Les commentaires multi-ligne
  - 3.5. Les boucles
    - 3.5.1. For loop
    - 3.5.2. While loop
  - 3.6. Les conditions
    - 3.6.1. Les comparaisons
    - 3.6.2. If
    - 3.6.3. Elif
    - 3.5.4. Else
    - 3.5.5. Match
  - 3.7. Les fonctions
  - 3.8. Les classes
    - 3.8.1. La programmation orientée objet
    - 3.8.2. L'initialisation
    - 3.8.3. Les bases sur les classes
      - 3.8.3.1 Les getter
      - 3.8.3.1 Les setter
      - 3.8.3.1 Les *private* variables
- **Projet : créé une carte d'identité**

4. Python dans son ensemble

- 4.1. L'importation de libraries
- 4.3. L'aléatoire
- 4.4. Ouvrir et fermer des fichiers
- 4.5. Les dictionnaires et JSON
- 4.6. Les Listes
- 4.7. Les classes : avancé
  - 4.7.1. Inhéritance
  - 4.7.2. Encapsulation
  - 4.7.3. Polymorphisme
- **Projet : créer un fichier d'identité avancé**

5. Python avancé : plongé dans le threading, l'asynchrone et le web

- 5.1. Le front-end vs. back-end

- 5.2. Les REST API
- 5.3. Les bases de données SQL
- 5.4. Les bases de données no-SQL
- 5.5. Envoyer et recevoir des informations avec les websockets
- 5.6. Faire un serveur API
- 5.7. Utiliser une API tierce
- 5.8. Threader un programme
- **Projet : faire un site de création et distribution de fiche d'identité**

---

1. INTRODUCTION ET CONCEPTS

1.1. Fonctionnement d'un ordinateur.

Les composants

Un ordinateur est une machine complexe, qui, a l'aide de calculs, nous aident au quotidien. Que ce soit au travaille, a l'école, à la bibliothèque, ou que vous vouliez acheter un article en ligne, ou même regarder une vidéo, vous utiliserez un ordinnateur.

Pour commencer, parlons des composants : * CPU : *Central Processing Unit*, c'est l'élément principal qui va nous permettre de faire ces calculs. Elle est composée d'une `horloge` qui est définie en *Hz*, et qui définie le nombre d'actions réalisé/réalisable par seconde. Il est aussi composé de `coeurs`, qui vont être cadencé et ordonné par l'`horloge`, qui effectueront leurs tâches associées. Les `threads` peuvent être représenté comme une boite, qui sera dépendante d'un `coeur`, qui effectuera le travaille envoyé par l'`horloge` a son coeur associé.

- GPU : *Graphics Processing Unit*, celle-ci n'est pas obligatoire, mais généralement nécessaire pour les ordinateurs de travail ou autre. Elle permet de faire de nombreux calculs en temps réel simultanément. Elle fonctionne en couche appelée `layer`, empilé les une sur les autres, commandé par de complexe calculs de vecteur et de matrices.
- RAM : *Random Access Memory*, un élément très important dans la programmation, car c'est généralement à l'utilisateur de décider de comment l'utilisé, en revanche, cela est moins important en Python. Elle est stock les données de manière temporaire, par exemple, les valeurs d'une variable dans un programme. Elle est différente du stockage physique (disque dur, etc.), parce qu'elle permet d'accéder très rapidement à une valeur, mais a pour inconvénient d'être réinitialisé lorsqu'elle n'est plus alimenté.
- Carte Mère : *MotherBoard*, c'est le composant qui relie tous ceux précédents entre eux. Elle n'a pas tant de fonctionnalité en elle-même, mais peut tout de même être couplé a une carte wifi par exemple, qui captera les réseaux wifis, et d'autre encore.

Mémoire RAM

La mémoire RAM est composé de 2 éléments pour chaque emplacement : l'`adresse` et la `valeur`. En réalité, dans beaucoup de language de programmation, si ce n'est tous, la mémoire utilisé est une `mémoire virtuelle`, qui sera entièrement gérée par le système d'exploitation; celui-ci va alors décider de l'organisation dans la `mémoire physique`. Ce processus permet d'éviter les erreurs de mémoire (exemple : 2 programme qui veulent accéder a la même adresse), et donc ajouté une couche de protection pour l'utilisateur et les applications elles même.

Prenons ce tableau en exemple :

Adresse Virtuelle	Adresse Physique	Valeur	Valeur réel
0x73B1FF	0x093F82	73	01001001
0x73B200	0x2F783A	`s`	01110011
0x73B201	0xB3E012	True	00000001

Voici ce comment on accèderait a une valeur :

```
valeur(0x73B1FF) -> 73
```

Alors qu'en réalité :

```
valeur(0x73B1FF) -> physique(0x093F82) -> valeur(01001001) en chiffre -> 73
```

Processeur

Comme vu précédemment, le processeur est constitué d'une horloge et de coeurs, qui sont eux même composé de threads. On peut imaginer un processeur tel une entreprise : * L'horloge est comme une horloge dans une entreprise qui définit le rythme de travail, mais ne prend pas de décisions. * Les coeurs sont des employés capables d'exécuter des tâches de manière indépendante, et c'est le système d'exploitation (un manager) qui leur assigne des tâches. * Les threads sont des sous-équipes au sein de chaque employé (coeur), qui divisent les tâches pour être effectuées simultanément dans un même coeur.

Pour résumer, voici l'arbre hiérarchique du processeur :

```
Horloge --- rythme de travail ---> coeur --- instructions ---> threads (si Hyper-Threading)
```

Threads Les threads sont souvent utilisés dans des applications ou projets à charge lourde, qui ont un besoin critique de performance ou de parallélisme dans les taches effectuées (simultanées). En programmation, on affecte fréquemment des fonctions complexes ou longue aux threads, afin de ne pas ralentir le programme entier. Par exemple, en Python :

Considérons cette fonction

```python
def prime_numbers(n: int, start: int = 2) -> list[int]:
    # éviter les erreurs de paramètres
    if n <= 1:
        print("Erreur, chiffre trop petit")
        return []
    if start < 2:
        print("Erreur, début trop petit")
        return []

    # le "coeur" de la fonction
    result: list[int] = [] # le resultat qu'on renverra
    for i in range(start, n):
        for j in range(start, i):
            if (i % j) == 0: # si le nombre est un multiple entier autre que 1 ou le même
                break
        else:
            result.append(i) # ajouter ce nombre au resultat
    return result
```

Cette fonction renvoi une liste des nombres premiers compris entre `start` et `n`.
Elle n'est pas optimisé, et peut prendre beaucoup de temps a éxecuter lors de paramètre plus élevés.

```python
import time

goal: int = 100_000 # le nombre maximal

curr = time.time() # le temps avant le lancement des fonctions
prime_numbers(goal) # les nombres premier compris entre 2 et 100,000
print(time.time() - curr) # affiche le temps actuelle - celui avant la fonction afin de déte
```

```
17.05051851272583
```

Soit 17 secondes pour trouver tous les nombres premiers de 2 à 100,000.

Maintenant, éxecutons cette fonction 2 fois, simultanément.

```python
import time
import threading

goal: int = 100_000 # le nombre maximal

thread1 = threading.Thread(target=prime_numbers, args=(goal//2, 2)) # le 1er thread executer
thread2 = threading.Thread(target=prime_numbers, args=(goal - goal//2, goal//2)) # le deuxi

curr = time.time() # le temps avant le lancement des fonctions

# lancer les threads
thread1.start()
```

```
thread2.start()

# attendre la fin de leurs opérations
thread1.join()
thread2.join()

print(time.time() - curr) # le temps entre le lancement des fonctions et leurs fin
```

```
4.005001544952393
```

Comme on peut le constater, le fait d'avoir réparti cette fonction sur 2 threads simultanés donne un résultat bien plus rapidement que sans le multithreading.

Le code ci-dessus est une illustration seulement, il n'est pas nécessaire de comprendre entièrement le code, mais uniquement de comprendre l'importance de l'utilisation de threads.

Cache

Le cache constitue la mémoire inclut directement dans le processeur. Du fait de sa proximité avec les coeurs, il est bien plus rapide que la RAM. Les données dessus ne sont rarement chargé plus de quelques millisecondes, voire quelques microsecondes. C'est ici qu'atterrissent les données de la RAM demandée par un programme, telle une sorte de "fil d'attente".

Carte Graphique

La carte graphique est un composant essentiel, et fréquemment utilisé, mais n'est pas indispensable. Elle permet d'afficher l'interface de l'ordinateur, dans le plus courant des cas. Elle est aussi utilisé pour sa puissance de calculs intense. Elle n'est pas requise dans les serveurs web par exemple, car seulement l'envoie et la réception de données est utile.

De plus en plus, les cartes graphiques sont considéré comme un des composants le plus important, principalement a cause du secteur du jeu vidéo ou encore de la crypto-monnaie.

Un GPU en lui-même est une sorte d'ordinateur compacte, en effet, elle possède une carte principale, similaire a une carte mère, BEAUCOUP de coeurs (2560 pour une NVIDIA GTX 1080), tels les processeurs, et une mémoire. Elles sont designé pour le calcul intensif simultané, grace a leurs milliers de coeurs, malgré leurs faibles performances.

Traitement des données

Généralement, les données utilisées par la carte graphique sont fait de matrices :

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Telle celle-ci dessus, et peut subir de complexe calculs comme la rotation, addition, multiplication, et division de matrices, ou encore même des calculs de vecteurs.

1.2. Python et ses applications

Les domaines adaptés

Le langage Python est très populaire dans de multiples secteurs, et continue de croitre au fil des années. Du fait de sa simplicité, popularité et de sa grande diversité de librairies, il est commun de retrouver Python dans : * l'IA et la science des données : les libraries d'IA et de *Machine Learning* (a.k.a. l'apprentissage des machines) sont faites en C++ du fait de sa rapidité, mais son ré-adapté en Python, et compose la plupart des grandes IA tel que Copilot ou ChatGPT. * le web, plus spécifiquement le *backend* soit le côté serveur. Il est utilisé pour des sites modérément grands, et permet une approche simple et rapide de la conception d'un serveur web. * l'apprentissage, beaucoup d'école (comme la France) apprennent aux élèves les principes de l'algorithmie via Python, grace a sa simplicité et sa syntaxe très "humaine". * les applications, de taille petite pour la plupart, tel qu'un générateur de mot de passe, ou un *pierre, feuille, ciseaux*.

Les domaines non-adaptés

Cependant, Python a tendance à être lent comparé à d'autre langage tel que *Java*, *C*, *C++* ou *Go*. Cette grande faiblesse fait que Python est moins adapté dans : * le secteur du jeu vidéo. Les jeux vidéos en 3D demande beaucoup de puissance de calcul afin de générer la lumière, sa refraction, la physique, etc. * les applications hautes performances, telles que les applications de montages photo et vidéo. En revanche, certaines parties de ces applications sont codé en Python, car elle ne nécessite pas de puissance de calcul.

1.3. Python et son environnement

Le Package Manager

Le package manager est un terme anglais, qui désigne un outil permettant entre autre l'installation de bibliothèque ou encore de mettre en place un environnement local au projet. Le package manager officiel de Python s'appelle `pip`. L'installation de `packets` (des bibliothèques) nécessite une connexion internet, et ont plusieurs répertoires d'installation par défaut. Il est possible de créer un environnement local dans notre projet, ce qui permettra à nos libraries d'être utilisable seulement par ce projet. Ces packets (ou *packages*) sont fait en Python, et permettent au développeur d'utiliser des fonctions ou classe déjà existante.

Plusieurs moyens existent afin d'utiliser `pip`, voici une liste complète :

```
pip <...args>
pip3 <...args>
python -m pip <...args>
python -m pip3 <...args>
```

> [i] Vérifiez bien que Python et (si possible) `pip` soit dans les variables d'environnement système, ou `%PATH%`, afin que la commande puisse être utilisée. Vous pouvez aussi utiliser la commande directement à partir de l'éxecutable `pip` à partir de son répertoire comme ceci : `<chemin_vers_pip> <commande_pip>`, par exemple `"C:\python312\Scripts\pip.exe" pip --version`.

Les librairies

Les libraries standards Python à par défaut une vingtaine de `package`, qui rassemble un large éventail de fonctionnalités. Dans la liste officiel des `Standard Library` de python, les plus populaires sont : * `datetime` : manipulation de dates. * `json` : manipulation d'objets et fichier json. * `asyncio` : fonctions asynchrone et autre. * `threading` : pour le multi-threading. * `typing` : pour la gestion de type, et la programmation statique. * `math` : pour les fonctions mathématiques telles que la racine carrée. * `requests` : recevoir et envoyer des données par HTTP. * `hashlib` : fonctions de hashing. * `os` : fonctionnalités du système d'exploitation. * `sys` : fonctionnalité du système. * `random` : pour les opérations aléatoires. * `sockets` : pour la gestion de socket web.

Les libraries populaires Les librairies externes les plus courantes dépendent de leurs cas d'utilisation : * REST API et serveurs Web : * `Flask` : une bibliothèque solide pour les projets de petite a grande échelle. * `Django` : spécifiquement prévu pour les plus gros projets, ayant des règles plus strictes et complexes. * `FastAPI` : adapté à de petits projets, son manque de fonctionnalités est compensé par ses hautes performances. * IA et Science des Données : * `numpy` : large librarie permettant de faire des array à deux dimensions, des formules de maths complexes, et autre. (est rarement utilisé seul). * `pytorch` : bibliothèque pour le *Machine Learning*. Accessible au débutant, mais tout de même puissant. * `tensorflow` : équivalent de `pytorch` mais plus complex et professionnel. * `scipy` : complète `numpy`. * Bases de données * `sqlite3` : pour SQLite. * `psycopg2` : pour PostGreSQL. * `pymysql` : pour MySQL. * `sqlalchemy` : une grande variété de base de données SQL. * `redis` : pour Redis (noSQL). * `pymongo` : pour MongoDB. * Les jeux * `pygame` : le plus populaire, permet des jeux simples. * `pyopengl` : utilise l'API d'OpenGL et se concentre sur les objets 3D.

Conda

Conda est un `interpreter` Python qui cible les développeurs du secteur de l'Intelligence Artificiel. Il vient avec son propre environnement appelé `Conda env`, et à de nombreuses librairies installées par défaut. Il possède une gestion des fichiers hors normes qui isole les projets les uns des autres, et réduit radicalement les erreurs de version d'outils entre plusieurs projets. Conda amène un certain nombre d'outils, mais en plus supporte plusieurs langages de programmation telle que *R*, *C++*, et bien plus encore.

1.4. Langage interprété vs. compilé vs. Byte-code

Langage Interprété

Un langage de programmation interprété est un langage qui ne subit pas de transformation entre le code source et la sortie. Ainsi, il est dépendant de l'`interpreter`, qui va, comme son nom l'indique, interpréter le code et l'exécuté.

Pour imager, prenons un message à faire passer à quelqu'un.

Imagine un ouvrier qui construit une maison sans plan complet. Il reçoit les instructions au fur et à mesure, comme quelqu'un lui dictant chaque étape sur place. Il pose une brique, puis attend la prochaine instruction. C'est ainsi qu'un langage interprété fonctionne : chaque ligne de code est exécutée immédiatement après avoir été lue, sans plan global préétabli.

L'avantage des langages de programmation interprété se résume généralement sur le confort d'utilisation. Ceux-ci ont tendance à être plus simple à apprendre et à manier, mais sont plus lent à execute ou à `debugger` car le programme est exécuté au fur et à mesure, sans savoir l'instruction suivante. Il ce peut donc que votre programme ait une erreur après plusieurs minutes de lancement, car une instruction n'avait pas été atteinte jusqu'alors. A noté que ces langages ont un accès très limité sur l'infrastructure de l'ordinateur, puisqu'une majeure partie est gérée par l'interpreter. Cela aussi fait d'eux des langages lents et peu performants à cause du passage à travers l'interpreter.

> [i] *Javascript*, *PHP*, et *Ruby* sont tous les trois des langages de programmation interprétée.

Langage compilé

Les langages compilés passent à travers un processus en plusieurs étapes, qui va décortiquer le code et le transformer en binaire, puis en executable. Cette étape est nécessaire seulement après les modifications du fichier source. Les fichiers compilés sont directement compréhensible pour la machine, cela permet donc de directement communiquer avec elle, sans passer à travers un programme quelconque. Les langages compilés sont ainsi bien plus rapide comparé aux autres languages, mais ont la spécificité de demander plus d'effort du développeur à cause de la gestion de la RAM et des types de variables.

Pour imager ce concept, illustrons par cette analogie :

Ici, un architecte prépare d'abord tous les plans de la maison. Il dessine chaque détail sur une feuille. Ensuite, une fois que tout est prêt, les ouvriers suivent ces plans pour construire la maison en une seule fois. Dans un langage compilé, le code est d'abord traduit entièrement en langage machine (les plans complets), puis exécuté par l'ordinateur.

> [i] Beaucoup de langages de programmation sont compilé, on peut y retrouver notamment *C*, *Rust* et *GoLang*.

Langage byte-code

Le byte-code représente une étape intermédiaire d'un langage compilé : le code source est analysé et transcrit en *byte-code*, qui ne peut pas être éxecuté ou compris directement par l'ordinateur. Un programme sous le nom de Virtual Machine, viendra interpréter ces fichiers. On peut considérer ce système comme un entre-deux de ceux présenter précédemment, car il est à la fois compilé et interprété.

L'architecte crée un plan général dans une sorte de croquis standardisé, mais ce croquis doit être interprété différemment en fonction des outils et des matériaux disponibles. Avant de commencer la construction, les ouvriers utilisent un manuel spécialisé pour comprendre et adapter ce plan intermédiaire à leurs outils spécifiques. C'est ainsi que fonctionne le bytecode : le code est traduit en une forme intermédiaire, puis un interprète (machine virtuelle) le traduit pour l'ordinateur spécifique.

> [i] Peu de langages de programmation utilisent cette technique, et les plus populaires sont *Java* et *C# (à prononcer "C Sharp")*.

Comparaison

	Interprété	Compilé	Byte-Code
Vitesse / 10	1	10	7
___Cross plat-form*___	Oui, mais a besoin d'un OS et de l'interpreter.	Dépend seulement du processeur.	Besoin de la Virtual Machine
Simplicité / 10	1	~8	~8
___Taille / 10**___	3	1	8
___Low Level / 10***___	2	10	9

* le *cross platform* désigne la capacité d'un programme d'être compatible sur différente machine (téléphone, ordinateur Windows, Mac, etc.).

** désigne la taille des fichiers produits. (plus petit est meilleur)

*** fait référence à la distance d'un programme à l'ordinateur : plus un programme est *low level*, plus il aura de contrôle sur la machine.

1.5. Types statiques vs. Types dynamiques

Ces deux mots désignent un style de programmation, qui peut être employé seulement avec les langages qui supportent la définition des types en temps réel. Le concept consiste à préciser le type de chaque variable, afin d'éviter les erreurs relaté à la mauvaise assignation de type, etc. Par exemple : `variable1 = 123`, reviendrait à `variable1 : nombre = 123`.

> [i] Les langages tels que *Python*, *TypeScript*, *JavaScript* et *PHP* supporte les 2 styles.

Types statiques

Les types statiques, sont des types déterminés avant même l'éxecution du programme. Ceci permet à éviter les erreurs de type, et de mieux manipulé celle-ci. En Python, pour déclarer un type, nous devons lui assigner un nom ainsi que sa valeur.

```
# <variable> : <type> = <valeur>
variable: str = "salut"
```

Ici par exemple, nous assignons à la variable `variable` le type `string`, soit une chaine de caractères, une phrase, représenté en python par le mot `str`. En revanche, cela peut causer des problèmes si l'on veut assigner cette variable à un type `NULL`, ou `None` en Python, c'est-à-dire RIEN. Pour y remédié, on spécifie que le type peut aussi être `None`, comme cela :

```
variable: str or None = "salut"
variable = None # ça marche
```

Types dynamiques

Les types dynamiques consistent seulement à ne pas préciser le type de la variable, et peut mener à des erreurs si l'utilisateur ne fait pas attention.

```
variable = "salut"
variable = 123
variable = [1, 2, 3]
```

Ce code ci-dessus marchera sans problème.

Conclusion sur les types

Il est préférable de coder avec des types statiques, pour éviter tous conflits, et améliorer la compréhension du programme en son ensemble.

1.6. L'algorithmie et les mathématiques dans la programmation

La programmation, historiquement, ainsi que tout ce qui concerne l'informatique, viens des mathématiques. Seulement, les ordinateurs ne calculs pas comme nous : nous utilisons des ensembles de chiffres, compris dans la base de 10, et utilisons notre cerveau, un outil puissant, mais complexe. L'ordinateur, lui, utilise du million d'interrupteurs ON & OFF, représenté en binaire 1 et 0, qui, lorsqu'ils sont assemblé, peuvent former des chiffres extrêmement longs et précis.

La compréhension du fonctionnement d'un ordinateur est importante pour les développeurs, bien que quelques langages soient épargnés ; Python en fait partit.

Les maths, dans leur globalité, est nécessaire, car, grace au développement de la logique, résolution de problème, et algorithmie, une grande partie des bases sont acquises, et permettent un apprentissage rapide et fluide. Prenons les boucles en exemple :

```
de 1 à 7, faire ...
```

qui équivaut à :

```
for i in range(1, 7):
    ...
```

peut être représenté par :

$$\sum_{i=1}^{7}(...) = (...)$$

Les mathématiques avancées sont requis pour les développeurs avancés, et ne concernent pas l'audience de ce livre, mais il est tout de même important a noté que la logique est surement la compétence **la plus importante** en programmation.

1.7. Les conventions

Les conventions sont une grosse partie des langages de programmation. Elles définissent des règles à suivre, pour homogénéiser le code des différents développeurs. On peut rapporter ce principe à celui des Sciences, qui définit les notations, formules et unitées international, tel que $km.h^{-1}$, ou encore $\sqrt{(a^2 + b^2)}$.

Il existe plusieurs types de notations : * *UpperCamelCase* : consiste à mettre en majuscule la première lettre de chaque mot, sans les séparer par un espace. (`BonjourLesAmis`) * *lowerCamelCase* : consiste à mettre en majuscule la première lettre de chaque mot, excepté le tout premier, sans les séparer par un espace. (`bonjourLesAmis`) * *snake_case* : consiste en remplacé les espaces par un tiret du bas `_`. (`bonjour_les_amis`) * *flatcase* : consiste a écrire tout en minuscule sans espace. (`bonjourlesamis`)

Python utilise le *snake_case*, et donc, chaque variable, fonction, à l'exception des classes qui utilisent *UpperCamelCase*, doivent se plier à ces conventions Les espaces étant interdit dans ~99% des langages, dans la nomination, il faut pouvoir distinguer les mots entre eux. Exemple en, Python :

```
phrase_de_bonjour = "salut"
def dire_bonjour(): ...
class Mot_Bonjour: ...
class MotBonjour: ...
```

Tout mot défini par l'utilisateur tel que la nomination des variables, classes, et fonctions, ont interdiction de commencé par un chiffre, de contenir des espaces ou caractère spéciaux tel que les accents, les symboles monétaires, etc.

---

2. Les outils

2.1. Les éditeurs de texte

Les éditeurs de texte sont présents sur quasiment n'importe quel ordinateur, et peuvent être très simpliste (NotePad sur windows), ou avancé, adapté pour la programmation avec des fonctionnalités supplémentaires. Pour la plupart des éditeurs avancés, une coloration de la syntaxe aide les développeurs à mieux lire et analisé leur code, et peut avoir accès à un correcteur, qui détecte et corrige les erreurs.

Beaucoup d'entre eux sont open-source, gratuit et léger (ne prend pas de place sur le disque).

Les plus populaires sont : * VSCode (*Visual Studio Code*) : un éditeur complet et open source créée par Microsoft, qui dispose d'un large panneau de personnalisation. * Sublime Text : éditeur de texte assez vieux, similaire a vscode mais moins avancé. * Brackets : principalement centré sur la programmation web, celui-ci est innovant et simple. * Atom : concurrent a VSCode

Je recommande tout de même l'utilisation de Visual Studio Code, car il possède généralement plus de fonctionnalité, de personnalisation et de *plugins* que les autres.

2.2. Les IDEs

Les IDEs sont des applications à part entière, qui complète les éditeurs de texte, et sont souvent spécifique à un seul langage. La majeure différence est qu'un IDE est une sorte de pack, qui contient tous les outils nécessaires pour le langage ciblé : linter (couleur de texte), debugger (corrige les bugs), le compiler / interpreter / VM. Il est autant adapté aux débutants, qu'au professionnel, par sa simplicité et son avancée. Il est à noter que beaucoup d'entre eux sont payants, les plus connus sont : * Visual Studio : similaire a VSCode, mais à la faculté de supporter la plupart des langages et leurs outils. (Gratuit) * Suite JetBrains : les IDEs JetBrains cible généralement 1 seul langage à la fois. Ils sont plus utilisé au niveau professionnel. (Gratuit & Payant) * Eclipse : IDE gratuit pour Java.

2.3. Les debuggers

Les debugger, comme son nom l'indique, est un outil qui aide les développeurs à chercher les erreurs dans le code, et possède de puissantes fonctionnalités telles que l'analyse de la mémoire, analysé une ligne spécifique ou encore découpé le code en petites étapes.

Tous les langages ne possèdent pas forcément un debugger, mais ils existent très souvent en open source, non officiellement rattaché au langage.

2.4. Environnement Python

Concept de l'environnement virtuel

Le principe d'un environnement virtuel en programmation est d'isoler le projet du reste du système, afin d'éviter les conflits de packet, et permettent souvent d'être reproduit, et donc faciliter l'échange de projet. Ils servent aussi à controlée les versions des librairies, des paramètres du projet, et plein d'autre encore.

Le compiler / interpreter sera généralement copié dans ce même environnement, mais sans l'entièreté des fichiers, afin de garder les bibliothèques importante.

Venv

`venv` est le gestionnaire d'environnement par défaut depuis Python 3.x, remplaçant le `pipenv` et `virtualenv` de Python 2.x. Il est amplement suffisant pour la plupart des projets, et évite aussi l'installation d'un environnement tiers.

Pour l'utilisé dans un projet, il suffit de taper dans l'invite de commande :

```
python -m venv <chemin vers le nouvel environnement>
```

Le nom le plus commun d'un environnement en python est `.venv`, qui se situera dans les fichiers *racine* du projet, c'est-à-dire le dossier contenant le projet entier.

Pour activer l'environnement, il faudra lancer un script :

```
./venv/Scripts/activate
```

ou encore

```
./venv/Scripts/activate.bat
```

Lors de cette action, vous verrez :

```
<nom de l'environnement> <chemin vers le project>
```

qui par exemple ressemblerait à :

```
(.venv) C:\Users\myuser\Documents\python_project
```

Lorsque vous êtes dans l'environnement, vous pouvez installer des *packages* a votre gout avec l'outil `pip` vu précédement.

```
pip install monsuperpackage
```

Conda env

L'environnement Conda est inclus avec leur interpreter Python, et est considéré comme avancé. Celui-ci dans un premier temps gère plusieurs langages différents, tels que *R* et *Python* dans le même projet. De plus, les environnements ne dépendent pas directement du projet, mais sont créé dans le répertoire de Conda lui-même, et permettent d'être partagé avec d'autre projet.

Afin de mettre en place un environnement Conda, il faut utiliser l'invite de commandes :

```
conda create --name <nom de l'environnement>
```

Il est aussi possible de spécifier la version de Python à utiliser :

```
conda create -n <nom de l'environnement> python=<version (3.11 par exemple)>
```

Pour activer l'environnement, similairement a `venv` :

```
conda activate <nom de l'environnement>
```

et l'installation des packages peut être au choix en utilisant pip ou conda:

```
conda install <nom du packet>
```

ou

```
pip install <nom du packet>
```

Poetry

Poetry est un projet open-source, avec un large éventail de fonctionnalités. Pour son installation, veuillez suivre le guide officiel. Afin de créer un projet avec son environnement inclut :

```
poetry new <nom de l'environnement>
```

Cette commande générera des fichiers de base, en plus d'un fichier `pyproject.toml`, qui contiendra toutes les informations du projet actuel.

Pour les projets éxistant :

```
poetry init
```

Permet de sélectionner le chemin actuel du terminal en tant que projet.

Pour l'activer :

```
poetry shell
```

Et pour installer des packets :

```
poetry install <nom du packet>
```

2.5. Le package manager

Le package manager est l'outil qui va installer les librairies, en choisissant leurs versions et autre. L'outil officiel de Python s'appelle `pip` et possède de nombreuses fonctionnalités.

Pour installer un package, on utilise la commande

```
pip install <package>
```

Et si l'on veut spécifier une version :

```
pip install <package>==<version>
```

Par exemple :

```
pip install pandas # installe pandas
pip install pandas==2.2.2 # installe pandas version 2.2.2
```

Il est aussi possible de lister les packages en dehors de l'environnement, avec un fichier `requirements.txt`. Pour y ajouter nos libraries, nous devons rentrer la liste de celles-ci dans le fichier `.txt`, comme ceci :

```
pip freeze > requirements.txt
```

Le terme `pip freeze` renverra une liste des packages installés, tandis que `... > requirements.txt` s'occupera d'enregistrer cette liste dans le fichier.

Afin d'installer la liste du `requirements.txt`, il suffit de taper :

```
pip install -r requirement.txt
```

---

3. Les bases

3.1. Les notations / particularitées syntaxique

Python est un langage de programmation avec une syntaxe simple, compréhensible et très proche de l'anglais. Il possède très peu de *keywords* : c'est-à-dire les mots prédéfinis. Celui-ci en possède 36, ce qui est relativement peu comparé à la plupart des autres langages tels que JavaScript qui en compte 63.

Python sépare les parties du code en utilisant des *indentations*, en appuyant sur `[TAB]`. L'indentation est généralement comprise entre 2 à 5 espaces, selon les applications et préférences. Elle définit des sortes de `scopes`, qui sont des parties independents de code, tel que le contenu d'une fonction, d'une classe, etc.

On notera que par exemple, ce code produira une erreur :

```
def hello(name: str) -> None:
print("hello, ", name, "!")
```

Le code considérera que la fonction `hello()` est vide, ce qui provoquera une erreur, et que, de plus, le paramètre `name` ne sera pas défini, car il appartient à la fonction elle-même.

En revanche :

```
def hello(name: str) -> None:
    print("hello, ", name, "!")
```

Ne produira pas d'erreur.

Cependant, il existe quelques exceptions, lors de l'accumulation de fonction, qui peuvent ressembler à un problème d'indentation, mais qui est courant afin de limiter la longueur des lignes.

Par exemple :

```
var = "Python"
var = (len(var.replace("P", "p")
        .split())
        .to_bytes())
```

Ce code est bon, car il utilise la fonction `split()` par-dessus `len(var.replace("P", "p")`, et `to_bytes()` enveloppant le tout.

Chaque caractère précédent le début d'un `scope`, doit, dans la plupart des circonstances, être `:`, tout comme les fonctions présentées précédemment ; cela est aussi applicable aux boucles et aux classes.

3.2. Les variables et ses types

Les variables en Python n'ont pas de préfix, et se déclarent directement par leurs noms : `variable = 123`. Elles peuvent être instanciées en avance, seulement en

leur déclarant une valeur, qui par défaut serait `None` :

```
text = None

...

text = "salut"
```

Les variables par défaut, dites primitives, sont moindres comparé à celles natives en C++ ou Java par exemple : * `string`, caractériser par le mot `str` en python, qui définit un ensemble de caractère, soit un texte par exemple. Il peut être défini entre des guillemets ou des apostrophes, `"` ou `'`. * `int`, qui définit tout nombre entier. * `float`, qui définit tout nombre à virgule ou / et entier. * `char`, qui définit un caractère, qui a une valeur numérique suivant les codes ASCII. * `None`, définit RIEN. * `bool`, qui est défini sur [0; 1], indiquant `True` or `False`.

Puis, pour rentrer dans les types plus complexes : `list`, qui définit une liste de plusieurs valeurs, tel que `[1, 2.321, "salut", False]` * `tuple`, similaire a une liste, mais non modifiable, ce qui est envoyé par défaut lors d'une fonction renvoyant plusieurs valeurs. * `dict`, un dictionnaire, fonctionnant sur le système `key->value`, c'est-à-dire un registre de recherche par nom. * `set`, qui peut entre considéré telle une liste, mais qui ne peu pas contenir une valeur en double.

Beaucoup d'autres types éxistes, mais sont plus occasionnels que les 10 vu ci-dessus.

Pour en assigner un a une variable, la déclaration est :

```
<variable>: <type> = <value>
```

Soit

```
name: str = "Clément"
```

Les noms de variable doivent pouvoir être comprise par tout le monde. Il faut donc leurs données un nom suffisamment descriptif, même abrévié.

3.3. Les fonctions de base

Gérer la console

La console est le principal outil du développeur. Elle fait partie intégrante du système et permet la réception et envoie de données à l'ordinateur.

Print `print()` est la principale fonction de Python pour afficher du texte dans le terminal. C'est dans ses paramètres (entre parenthèse) que nous insérons le texte à écrire, mais aussi ou nous avons des réglages basiques que nous pouvons modifier à notre guise.

Exemples :

```python
x: int = 2

print(x) # 2
print("salut") # salut
print("salut", x) # salut 1
print("salut" + str(x)) # salut1
```

Les valeurs passées dans print, peuvent être séparé par des virgules, ou des `+`. Les virgules mettront automatiquement un espace entre le texte précédent et le suivant, tandis que l'opérateur `+` les ajoutera à la suite.

> [i] Les valeurs ajoutées avec `+` doivent d'abord être converti en `string`, pour cela, il faut seulement faire `str(<valeur>)`.

Il existe un autre moyen d'intégrer une variable dans une chaine de caractères comme ceci :

```python
f"<texte>{<variable>} "
```

Le `f` devant les guillemets spécifie que le texte contient une variable, et les accolades `{}` spécifie qu'une variable se trouve à l'intérieur.

```python
x: int = 2

print(f"salut {x}") # salut 2
```

L'un des paramètres le plus utilisé est `end`, qui doit être spécifié qu'il est complémentaire, et ne fait pas partie du texte à afficher. Étant donné que `print()` par défaut insert un retour à la ligne, on peut forcer celui-ci à être différent. Par exemple sans le paramètre :

```python
print("Salut")
print("Rodrigo")
```

Résultat :

```
Salut
Rodrigo
```

Exemple avec le paramètre :

```python
print("Salut", end="")
print("Rodrigo", end="")
```

Résultat :

```
SalutRodrigo
```

Exemple avec un second paramètre :

```python
print("Salut", end=" ")
print("Rodrigo")
```

Résultat :

```
Salut Rodrigo
```

Input `input()` est une fonction pour récupérer ce que l'utilisateur entre dans sa console. Elle a pour seul paramètre un texte à afficher pour demander une donnée à rentrer. Cette fonction renvoie la réponse de l'utilisateur, et doit donc être assigné à une variable.

Exemple :

```
name: str = input("Entrez votre nom: ")
print(f"Bonjour, {name}!")
```

Résultat :

```
Entrez votre nom: Ethan
Bonjour, Ethan!
```

Les opérations et calculs

Les calculs en programmation sont souvent très similaire à ceux utilisés dans la vraie vie. Par exemple `1 + 1` marchera, ainsi que tout-autres opérations.

Il est juste requis de préciser le signe entre les variables, comme `3x` équivaut à `3*x`.

Les virgules sont avec des points donc `3,14` sera `3.14`.

Les puissances sont représentés par deux asterix tel ceci `2**2` (2 puissance 2).

La racine carrée en revanche est utilisable depuis la bibliothèque `math`, et s'écrit `math.sqrt()`, et le nombre entre ces parenthèses.

3.4. Les commentaires

Les commentaires dans le code sont des parties qui ne sont prises en compte lors du lancement d'un programme. Ils servent entre autre à donner des détails, préciser ou expliquer certaines parties du fichier. Il est courant, chez les professionnels notamment, lorsque leurs projets sont échangé entre eux.

Les commentaires en fin de ligne

Les commentaires en fin de ligne correspondent a un commentaire qui n'a pas de marqueur de fin, c'est-à-dire qui s'arrêtera automatiquement à la fin de la ligne.

```
# je suis un commentaire de fin de ligne
print("Hello world!") # je suis un 2eme commentaire de fin de ligne
```

Les commentaires multi-ligne

Les commentaires multi-ligne s'étendent, comme son nom l'indique, sur plusieurs lignes. Ils servent aussi aux `string` de plusieurs lignes.

```
'''
mon
commentaire
super
long
'''

print("Hello world!")

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

3.5. Les boucles

Les boucles sont un concept important dans la programmation. Elles contiennent a minima un paramètre. Nous allons voir les 2 sortes de boucles et comment les utilisées.

For loop

Les `for` loop sont des boucles qui contiennent une condition et une valeur. Elles ont une variable (la valeur), qui sera par défaut itéré jusqu'à atteindre la condition. On peut donc traduire en langage naturel une boucle `for` comme cela :

```
pour i, si i < 30, alors faire <...>, i = i + 1
```

La syntaxe de cette *loop* est:

```
for <variable> in range(<numero de fin>)
```

`<variable>` est égal à 0 par défaut, mais il est possible d'outrepasser celle-ci :

```
for <variable> in range(<valeur de debut>, <valeur a atteindre>, <valeur a ajouter a la vari
```

Donc :

```
for i in range(8):
    ...
```

Revient à faire

```
for i in range(0, 8, 1):
    ...
```

Il est à noter que les arguments dans `range()` ne sont pas obligatoire, et son remplit dans l'ordre. Par exemple, si on considère que `range()` a les paramètre `start`, `objective`, `step` et que `start` et `step` ont pour valeur par défaut `0` et `1` respectivement, si l'on écrit `for i in range(5, 10)`, alors `start` sera égal à `5`, `objective` égal à `10` et `step` gardera sa valeur par défaut.

Pour résumer les bases :

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

Affichera

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

> [i] `start` atteindra au maximum `objective-1`.

Il est aussi possible de `loop` à l'intérieur d'une variable. Pour cela, il faut remplacer `range()` par la variable. > [i] Certain type ne sont pas compatible avec les boucles.

```
for i in ["s", "a", "l", "u", "t"]:
    print(i)
```

Résultat :

```
s
a
l
u
t
```

Cela aurait aussi pu marcher avec une variable de type `str`, ou `tuple` par exemple.

> [i] Les variables par défaut utilisé dans les `for-loop` sont i, puis j, en deuxième. Il est possible de changer le nom de ces variables si besoin. Si la variable ne compte pas être utilisé, on l'appellera (par convention) `_`.

While loop

Les boucles `while`, contrairement au `for-loop`, ne possède seulement une condition : il faudra donc mettre cette condition à jour par nous même.

La syntaxe est la suivante :

```
while <condition>:
    ...
```

Si la condition n'est pas changé dans la boucle, une boucle infinie se produira : le seul moyen pour l'arrêter est de stopper le programme. L'usage correct est donc, par exemple :

```
i = 0
while i < 10:
    i+=1 # i = i + 1
```

Cela revient à faire une boucle `for`.

Pour montrer un exemple plus plausible :

```
user_input = input("Entrez votre prénom : ")

while user_input != "Clément":
    print("Tu n'est pas mon Administrateur !")
    user_input = input("Entrez votre prénom à nouveau : ")
```

Pour conclure, les boucles `for` sont utilisé lorsque l'on sait le nombre d'itérations à faire, tandis que les boucles `while` servent lorsque l'on ne connait pas le nombre d'itérations.

3.6. Les conditions

Les comparaisons

Il existe différents opérateurs pour comparer des valeurs entre elle : * `==` : `<val1> == <val2>`, renvoie `True` si les valeurs sont égales, sinon elle renvoie `False`. * `!=` : `<val1> != <val2>`, renvoie `True` si les valeurs ne sont pas égales, sinon elle renvoie `False`. * `<=` : `<val1> <= <val2>`, renvoie `True` si `val1` est plus petit ou égal à `val2`. * `>=` : `<val1> <= <val2>`, renvoie `True` si `val1` est plus grand ou égal à `val2`.

Il existe d'autre comparaison en Python, tel que `in`. Celle-ci sert à verifier si une valeur est dans une `list` ou dans un `string`.

If

`if` signifie `si` et s'utilise pour vérifier si un état est vrai ou faux, `True` ou `False`. La syntaxe est :

```
if <expression>:
    ...
```

`<expression>` ici va généralement correspondre au opérateur de comparaison vu dans la section précédente. Le code sous `if` s'exécutera seulement si `<expression>` est égale à `True`.

Voici quelques exemples d'utilisation :

```python
num: int = 3

if num == 3:
    print("num = 3") # cette partie s'exécutera
```

```python
num: int = 3

if num == 4:
    print("num = 3") # cette partie ne s'exécutera pas
```

```python
num: int = 3

if num != 4:
    print("num != 3") # cette partie s'exécutera
```

Comme les prochains mots-clé

Elif

`elif` viens OBLIGATOIREMENT a la suite d'un `if`. Ce terme est une abréviation de `else if` et signifie `sinon si`. Elle possède la même propriété que `if`, car elle aussi doit être accompagné d'une expression.

Voici quelques exemples d'utilisation :

```python
num: int = 3

if num == 3:
    print("num = 3") # cette partie s'exécutera
elif num == 4:
    print("num = 4")
```

```python
num: int = 3

if num == 4:
    print("num = 3")
elif num == 3:
    print("num = 4")  # cette partie s'exécutera
```

```python
num: int = 3

if num != 4:
    print("num != 4") # cette partie s'exécutera
elif num != 3: # cette partie sera sauter
    print("num != 3")
```

Else

`else` doit forcément être la fin d'une expression `if`, peu importe la présence de `elif`. `else` signifie `sinon`, et n'accepte pas d'expression. Elle désigne la fin de

la condition, si aucune des précédentes n'ont été validées.

Voici quelques exemples d'utilisation :

```
num: int = 3

if num == 3:
    print("num = 3") # cette partie s'exécutera
elif num == 4:
    print("num = 4")
else:
    print("num n'est ni égale à 3 ni à 4")
```

```
num: int = 3

if num == 4:
    print("num = 3")
else:
    print("num n'est pas 4")
```

```
num: int = 3

if num != 4:
    print("num != 4") # cette partie s'exécutera
else:
    print("num est possiblement égale a 4")
```

Match

Les `if` / `elif` / `else` sont une éxecution de condition hiérarchique. La première condition qui est validée empêche les prochaines de l'être. Par exemple :

```
name: str = "Jean Pierre Fred"

if "Jean" in name:
    print("Il s'appelle Jean")
elif "Pierre" in name:
    print("Il s'appelle Pierre")
elif "Fred" in name:
    print("Il s'appelle Fred")
else:
    print("Il n'a pas de nom")
```

Résultat :

```
Il s'appelle Jean
```

Comme on peut le voir, les conditions suivantes ont été ignorées.

Pour y remédier, nous pouvons diviser ces conditions en conditions individuelles :

```
name: str = "Jean Pierre Fred"

if "Jean" in name:
    print("Il s'appelle Jean")
if "Pierre" in name:
    print("Il s'appelle Pierre")
if "Fred" in name:
    print("Il s'appelle Fred")
```

Résultat :

```
Il s'appelle Jean
Il s'appelle Pierre
Il s'appelle Fred
```

Depuis Python 3.10, une implémentation de `match`, qui simplifie l'écriture des `if...elif...else` :

```
match <variable>:
    case <valeur1>:
        print(<valeur1>)
    case <valeur2>:
        print(<valeur2>)
    ...
```

Le mot-clé `match` expose la valeur de la variable qui suit, et le keyword `case` vérifie si la valeur de la variable est égale à sa valeur confiée.

```
name: str = "Jean"

match name:
    case "Fred":
        print("Il s'appelle Fred")
    case "Jean":
        print("Il s'appelle Jean")
    case "Pierre":
        print("Il s'appelle Pierre")
    case _:
        print("nom inconnu")
```

Reviens à ceci :

```
name: str = "Jean"

if name == "Fred":
```

```
    print("Il s'appelle Fred")
elif name == "Jean":
    print("Il s'appelle Jean")
elif name == "Pierre":
    print("Il s'appelle Pierre")
else:
    print("nom inconnu")
```

Résultat :

```
Il s'appelle Jean
```

3.7. Les fonctions

Les fonctions permettent de diviser le code en plusieurs parties, et d'exécuter une action spécifique sans avoir besoin de répéter le code. On peut y passer des paramètres, qui auront un impacté dans cette même fonction.

La syntaxe est la suivante :

```
def <nom de la fonction>(<parametre>):
    <action du code>
```

Il est possible de renvoyer une valeur, avec le mot `return`, et la valeur. Prenons en exemple une fonction simple

```
def function():
    a = 2+2
    # a = 4

function() # appelle de la fonction
```

Mettre le nom de la fonction va l'*appeler*, et le code a l'intérieur sera exécuter. Ici, rien ne se passe. Seulement la variable `a` aura la valeur 4, uniquement pendant le temps d'exécution de la fonction.

```
variable = function() # variable = None
```

Ici aucune valeur n'est renvoyé, et la variable aura donc le type `None` par défaut.

En revanche, on peut renvoyer `a` afin de l'assigner correctement à cette variable.

```
def function():
    a = 2+2
    return a

function() # n'a pas d'effet
var = function() # 4
```

Voici des exemples de fonctions :

```python
def say_hello():
    print("Hello!")

say_hello()
```

Console :

```
Hello!
```

```python
def hello():
    return "Hello!"

print(hello())
```

Console :

```
Hello!
```

Afin de passer des paramètres, aussi appellés arguments, on peut les mettre dans les parenthèses, par leurs noms. On peut aussi leur assigner une valeur par défaut, qui changera seulement si l'utilisateur le décide.

> [i] Les paramètre par défaut doivent OBLIGATOIREMENT être déclaré à la suite des variables !

```python
def say_hello(name: str):
    print(f"Hello, {name}!")

# say_hello() -> erreur, car le paramètre n'est pas spécifié
say_hello("Clément")
```

Console :

```
Hello, Clément!
```

Ou encore, une fonction mathématique :

```python
def f(x: int):
    equation = 3 * x + 8
    return equation

print(f(0))
print(f(1))
```

Console :

```
8
11
```

Pour les arguments par défaut :

```python
### ERREUR
def say_hello(text: str = "Hello", name: str):
```

```
    print(text, name)
### ERREUR
```

Alors que

```
def say_hello( name: str, text: str = "Hello"):
    print(text, name)

say_hello("Clément")
```

Console :

```
Hello Clément
```

Et

```
def say_hello( name: str, text: str = "Hello"):
    print(text, name)

say_hello("Clément", "Hello Mr./Mrs.")
```

```
Hello Mr./Mrs. Clément
```

Pour définir un type à une fonction, de sorte que la valeur renvoyée soit du bon type, il suffit seulement de mettre `->` a la fin des parenthèses lors de la déclaration.

Exemple :

```
def say_hello(name: str) -> None:
    print(f"Hello {name}!")

say_hello("Clément")

# ne retourne rien, donc -> None

def say_hello(name: str) -> str:
    return f"Hello {name}!"

greeting: str = say_hello("Clément")
print(greeting)
# retourne un string, donc -> str
```

3.8 Les classes

La programmation orientée objet

Les classes, aussi appelées `object` ou objet en français, permette de réunir des variables et fonctions, et de créer un *plan* qui pourra être utilisé pour l'instancier de multiple fois. Elle possède un `constructeur`, qui sera activé dès l'assignation d'une classe, et définira un `état` (variables) ainsi qu'un `comportement` (fonctions).

On peut représenter un objet tel ceci, en langage naturel :

```
Chiens :
 - état :
  - age
  - race
 - comportement :
  - aboie
  - joue
```

Cela est une classe, par laquelle nous pouvons créer un objet `chien`.

```
chien = Chiens :
 - état :
  - age = 10
  - race = doberman

chien -> aboie
```

[i] Une classe sera automatiquement interprété en tant que type !

Les classes sont utilisées dans deux contextes différents : * L'organisation du code, impliquant la réutilisation de variables et le *wrapping* du code. * La réutilisation du code pour créer plusieurs entités par exemple.

Réutilisation (exemple dans le contexte d'un jeu vidéo) :

```
Ennemie :
 - pts_vie
 - dégats
 - mana
 - attaque()

spawn_ennemie (x) :
 - fait apparait x ennemies
```

Ici, la classe est réutilisé afin de créer plusieurs instances d'Ennemie, sans devoir écrire x ligne de code pour x ennemie.

Organisation (dans le même contexte)

```
Jeu :
 - joueur1
 - joueur2

 - start()
 - end()

jeu = Jeu
jeu -> start()
```

L'initialisation

Une classe contient par défaut une section d'*initialisation*, qui portera les variables de l'objet tout au long de son existence. Toutes variable et fonction accessible globalement par l'objet seront assigné par le mot clé `self`, définissant son appartenance.

Afin de déclarer une classe :

```
class <nom>:
    ...
```

Cette classe étant vide, il faut la compléter avec a minima l'initialisation :

```
class Person:
    def __init__(self):
        self.name = "Clément"
        self.age = 17
```

Ici, un objet de type `Person` sera instancier avec les valeurs par défaut de `name = "Clément"`, `age = 17`. On peut accéder à ces variables, en replacement le `self` par l'instance elle-même :

```
person = Person()
print(person.name) # Clément
```

Ces variables seront donc accessible à tout moment, à l'intérieur ainsi qu'à l'extérieur de l'objet.

Afin de définir ces variables au moment de l'instanciation, il faut ajouter les paramètre désirés dans `__init__`.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person() # erreur, name et age ne sont pas définit
person = Person("Clément", 17)
```

Nous pouvons aussi leur donner une valeur par défaut de la même manière que pour les fonctions.

Les bases sur les classes

Maintenant que nous savons créer la base d'une classe, nous pouvons y placer des fonctions.

Les getter Une fonction *get* sert à récupérer une variable. Contrairement à accéder directement à la variable, on peut y appliquer une seconde logique derrière, ainsi que "protéger" ces variables.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_name(self) -> str:
        return self.name
```

Ici, il n'y a pas de logique secondaire, mais cette fonction empêche tout de même de modifier une variable par erreur, car nous n'accédons pas directement à la variable, mais seulement à sa valeur qui a été copiée.

Incluons une condition, qui vérifiera l'age par exemple.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_age(self) -> int:
        if self.age < 1:
            return 1
        if self.age > 99:
            return 99
```

Maintenant, l'age affiché pourra seulement être compris entre 1 et 99.

Les setter Les *setters* complètent les *getters*, et vont directement modifier les variables. Tout comme précédemment, elles permettent d'inclure une seconde logique qui évitera les erreurs critiques.

Prenons par exemple une application très importante pour l'armée. Les soldats aimeraient déliverer un message codé, contenant seulement des chiffres. L'opérateur va donc leur envoyé un message en numéros. Malheureusement, l'application sur le serveur n'a pas été prévu pour envoyé des chiffres directement, mais un texte contenant des chiffres.

Ici, le serveur va s'arrêter avec une erreur et interrompra une information critique.

En revanche, s'il y avait un *setter*, qui préviendrait l'utilisateur qu'il doit l'envoyer sous format texte, a la récéption de sa requête, ou encore mieux, transformera automatiquement le message en texte de chiffre.

Prenons pour ce code l'exemple d'un jeu vidéo :

```python
class Player:
    def __init__(self):
        self.health = 100 # la vie du joueur
        self.damages = 3 # les dégats que le joueur fait
```

```
    def set_health(self, health):
        self.health = health

    def damage_player(self, damages):
        new_health = self.health - damages
        self.set_health(new_health)
```

Ci-dessus, la fonction `damage_player`, activé chaque fois que le joueur prend des dégâts, calculera la vie du joueur après s'être fait attaquer, et l'enregistrera grâce à la fonction `set_health`.

```
player = Player()
player.damage_player(1200)
print(player.health) # on n'a pas défini de getter pour l'exemple, mais pensez à les utilis
```

Résultat :

```
-1100
```

Le joueur à sa vie en négatif, ce qui est techniquement impossible. Pour cela, nous pouvons ajouter une condition :

```
...

def set_health(self, health):
    if health < 0:
        self.health = 0
    else:
        self.health = health

...
```

Ici la vie ne pourra plus descendre en négatif.

Les *private* variables Contrairement à la plupart des langages OOP (*Object Oriented Programming*), il n'est pas réellement possible de définir l'élément d'une classe comme un membre privé ou public. Un membre privé est une variable ou fonctions d'une classe qui ne peut être appelé en dehors d'elle-même.

Par exemple :

```
Chien:
public:
    get_identity():
        return "get_age() get_name()"
private:
    name
    age
    get_age()
    get_name()
```

```
chien = Chien
# ne marchera pas #
# chien.name
# chien.age
# chien.get_age()
# chien.get_name()

# marchera #
# chien.get_identity()
```

Le seul moyen “caché” de répliquer ce comportement est en plaçant `__` (deux tirets du bas) devant la déclaration :

```
class Chien:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def __get_name(self):
        return self.__name

    def __get_age(self):
        return self.__age

    def get_identity(self):
        return f"{self.__get_name()}, {self.__get_name()}"


chien = Chien("Freddy", 5)
chien.get_identity() # marche !
chien.__get_name() # ??? marche pas
```