**Documentation de mon code**

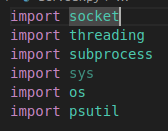
[**Serveur.py → page 1 à 4**](#mgnrm5nkzjoc)

[**ServeurEsclave.py → page 5 à 6**](#s8qrrygkxulz)

[**Client.py → page 7 à 12**](#g8mhhkkzqloq)

**Serveur.py**

**Les importations**



-J’importe toutes les “librarys” qui seront utiles à la réalisation de ce projet.

**socket** : permet de créer des sockets TCP/IP pour la communication entre le client et le serveur.

**threading** : permet de lancer des threads pour gérer plusieurs connexions lorsque plusieurs clients se connectent.

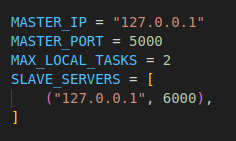
**subprocess** : On peut ’exécuter des commandes externes et capturer leur sortie

**sys** : fournit des fonctions et variables liées à l’interpréteur Python

**os** : permet d’interagir avec le système d’exploitation (ex. vérifier/supprimer des fichiers).

**psutil** : permet d’obtenir des informations sur les ressources système, comme la charge CPU.

**Les variables Globales**

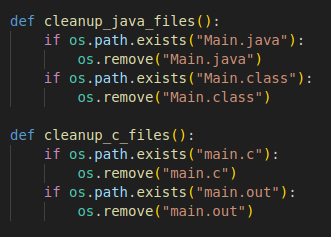


**MASTER\_IP** : l’adresse IP sur laquelle le serveur maître écoute les connexions.

**MASTER\_PORT** : le port sur lequel le serveur maître écoute (port 5000).

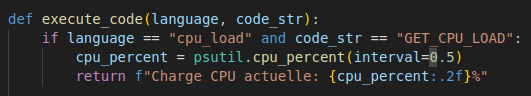
**MAX\_LOCAL\_TASKS** : le nombre maximum de tâches que le maître accepte d’exécuter localement en même temps. Au-delà de ce seuil, il délègue au Serveur Esclave..

**SLAVE\_SERVERS** : liste des serveurs esclaves. J’ai fait le choix d’implémenter un seul serveur esclave qui va écouter sur le port 6000.



Ces 2 classes suppriment les fichiers temporaires qui ont été créés lorsque des fichiers java ou C ont étés compilés.

**execute\_code**



J’utilise **psutil** pour obtenir le pourcentage d’utilisation de mon CPU sur mon interface graphique.

Le cœur de la fonction **execute\_code** consiste à exécuter du code dans un langage qu’on va pouvoir choisir sur l’interface graphique.

Voici les 3 langages de programmation que l’on peut sélectionner:

**Python →**

Il crée un processus python3 -c "code\_str" via subprocess.run().

Il capture la sortie standard et les erreurs.

**Java→**

Il écrit le code dans Main.java.

Il compile ensuite (javac Main.java).

Et il exécute (java Main).

Il va aussi nettoyer les fichiers grâce à **cleanup\_java\_files()** comme je l’ai expliqué précédemment.

**C →**

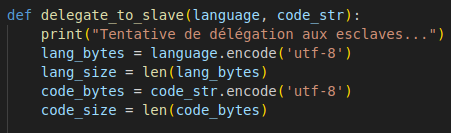
Il écrit le code dans main.c.

Il compile (gcc main.c -o main.out).

Il exécute (./main.out).

Il nettoie les fichiers grâce à **cleanup\_c\_files()** comme je l’ai expliqué avant.

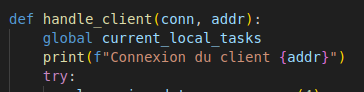
**delegate\_to\_slave**

****

Cette classe consiste à déléguer les tâches au serveur secondaires qui est mon Serveur Esclave.

Lorsque mon Serveur Maître reçoit plus de 2 codes à compiler en même temps, il va demander aux Serveur Esclave de gérer cette tâche.

**handle\_client**

****

Il y a plusieurs étapes dans cette classe.

Tout d’abord le serveur reçoit la taille du langage ainsi que la taille du code.

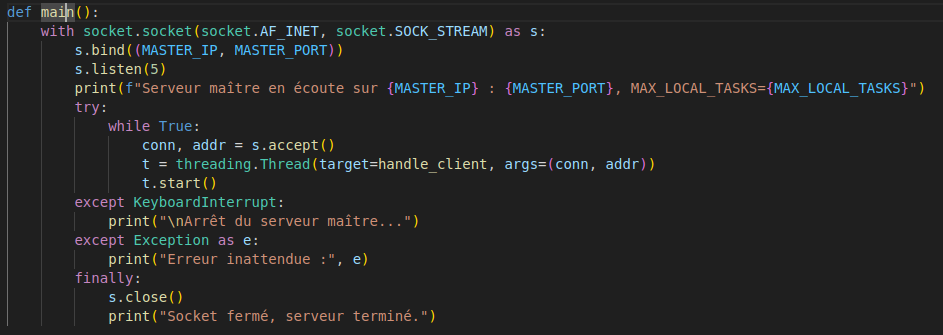
Il vérifie ensuite combien de tâches sont déjà en cours.

Si on dépasse les 2 tâches maximum, alors il délègue au Serveur Esclave.

Sinon il peut exécuter cette tâche localement.

Il renvoi ensuite la réponse au client.

**main**

****

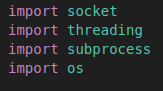
Cette classe crée une socket TCP et l’associe à l’@ ip et au port.

La boucle principale sert à accepter une nouvelle connexion client.

Pour finir, elle ferme la socket proprement avant de terminer.

**ServeurEsclave.py**

**Les importations**



**socket** : permet de créer des sockets TCP/IP pour la communication entre le client et le serveur.

**threading** : permet de lancer des threads pour gérer plusieurs connexions lorsque plusieurs clients se connectent.

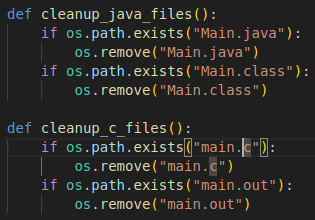
**subprocess** : On peut ’exécuter des commandes externes et capturer leur sortie

**os** : permet d’interagir avec le système d’exploitation (ex. vérifier/supprimer des fichiers).



Le ServeurEsclave écoute sur l’adresse ip 127.0.0.1 en localhost.

Il écoute sur le port 6000.



Comme sur le Serveur Maître, ces 2 classes suppriment les fichiers temporaires qui ont été créés lorsque des fichiers java ou C ont étés compilés.

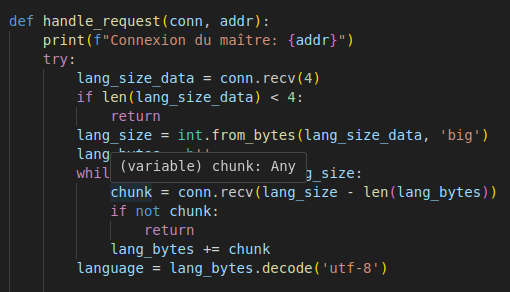
**execute\_code**

****

Cette classe reprend quasiment les mêmes fonctionnalités que la classe execute\_code sur le Serveur Maître à quelques exceptions près.

Ajout du TimeOut de 20 secondes. Si le code prend plus de 20 secondes à s'exécuter, une erreur de type “subprocess.TimeoutExpired”.

**handle\_client**

****

Cette classe gère une connexion entre le Serveur Maître et le Serveur Esclave.

Le Serveur Esclave reçoit le langage du fichier

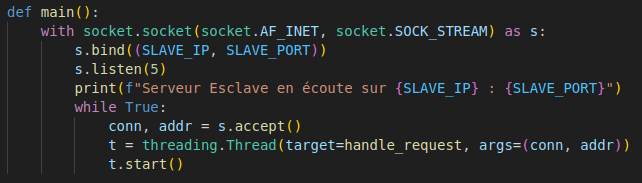
→ reconstruit une chaîne langage et source

→ affiche qu’il reçoit le code dans sa console

→ il compile et exécute le code

→ renvoi la réponse à l’interface graphique

**main**



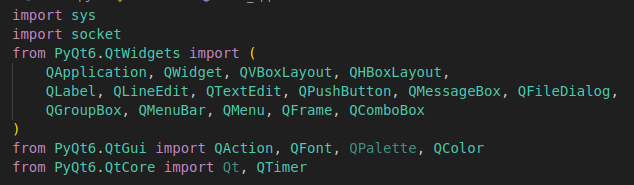
Comme pour la classe main dans le code du Serveur Maître, elle crée une socket TCP et l’associe à l’@ ip et au port.

La boucle principale sert à accepter une nouvelle connexion client.

Pour finir, elle ferme la socket proprement avant de terminer.

**Client.py**

**Les Importations**

****

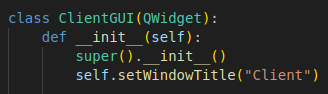
**sys** : fournit des fonctions et variables liées à l’interpréteur Python

**socket** : permet de créer des sockets TCP/IP pour la communication entre le client et le serveur.

**PyQt6** : fournit tous les widgets et les objets nécessaires pour construire l’interface graphique.

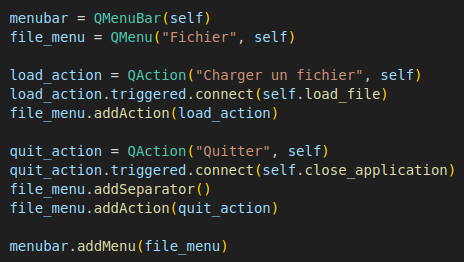
**QTimer** : sert à rafraîchir la charge CPU.

**ClientGUI**



Cette classe sert à initialiser l’interface graphique, initialiser les couleurs et créer la barre d’outils.

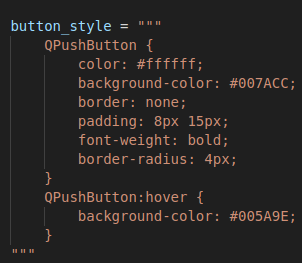
**Barre de menu**



Je peux charger un fichier à partir de mon pc.

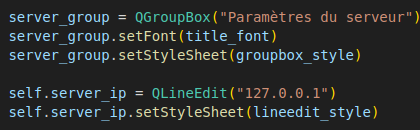
Je peux aussi quitter l’interface graphique proprement.

**Styles**



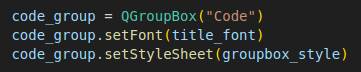
Apparence des boutons de l’interface graphique.

**Paramètres du Serveur**

****

Toute cette partie concerne la configuration du Serveur en commencant par renseigner son adresse IP pour pouvoir s’y connecter grâce à la ligne “self.server\_ip”.

**code\_group**

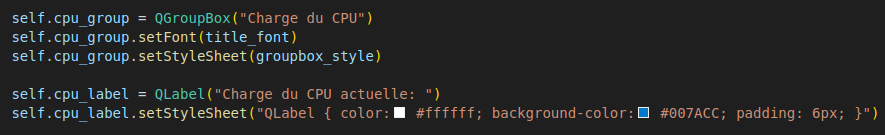


Gère tout ce qui concerne l'envoi et l’édition du code.

La zone de code est déjà remplie par le code “Hello World!”

Il y a aussi le bouton pour **Charger** un fichier (*self.load\_file*), pour **Envoyer** un fichier (*self.send\_code*), pour **Effacer** (*self.clear\_code*), et pour **Quitter** (*close\_application()*) l’interface graphique.

**self.cpu**

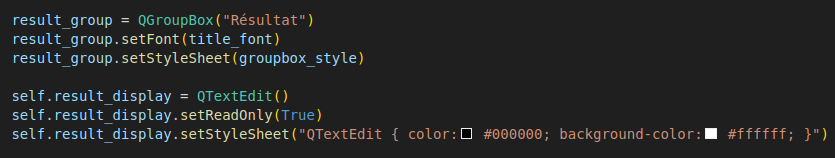


cou\_group sert à renvoyer la charge CPU grâce au serveur.

self.cpu\_label sert à afficher la phrase de mon choix.

self.timer sert à rafraîchir cette donnée toutes les 10 secondes (pour ne pas parasiter le CMD du Serveur)

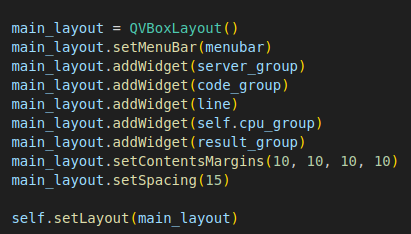
**result\_group**



result\_group sert à afficher les sorties ou les erreurs renvoyées par le serveur.

self.result\_display sert à afficher la réponse du serveur (en lecture seule)

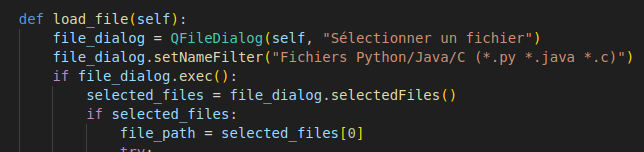
**main\_layout**



main\_layout sert à empiler les groupes dans la fenêtre

J’ajoute progressivement server\_group, code\_group, cpu\_group et result\_group.

**load\_file**

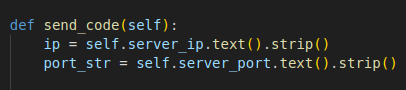


Cette fonction sert à ouvrir une boîte de dialogue pour sélectionner un fichier en .py, .java ou .c.

Ensuite, il lit le contenu du fichier et l’affiche dans self.code\_edit.

L’extension est automatiquement ajustée avec self.language\_combo.

**send\_code**

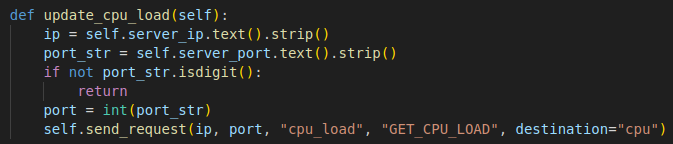


send\_code récupère l’IP et le port du serveur maître.

Il vérifie si le code n’est pas vide et récupère le langage sélectionné.

Appelle send\_request(...) pour envoyer le code au serveur.

**update\_cpu\_load**

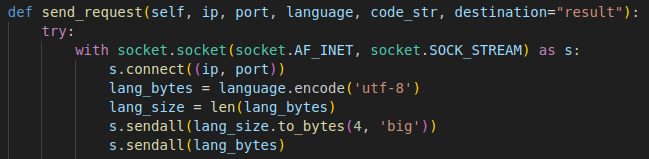


Cette commande est appelée automatiquement toutes les 10 secondes.

Elle envoie une requête spéciale (language="cpu\_load", code\_str="GET\_CPU\_LOAD") au serveur maître pour obtenir la charge CPU.

Indique la destination = "cpu" pour que la réponse soit affichée dans self.cpu\_label.

**send\_request**



send\_request gère la logique réseau →

Créer une socket TCP.

Se connecte au (ip, port).

Envoye la taille + le langage + le code (même protocole que le serveur).

Reçoit la taille de la réponse, puis la réponse en bytes.

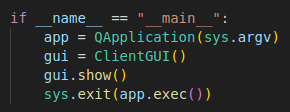
Convertie la réponse en chaîne

Si destination == "cpu", il met à jour self.cpu\_label. Sinon, il met à jour self.result\_display.

**close\_application**



Ferme proprement l’interface graphique en cliquant sur le bouton “Quitter”.



Crée une application grâce à PyQT6

Instancie ClientGUI

Affiche la fenêtre

Lance la boucle d’évênements