



Sommaire

## Introduction :

Dans le cadre du Master d'informatique de l'UPMC et au sein du parcours STL, un projet doit être réalisé lors du deuxième semestre de première année. Ainsi nous avons décidé de choisir le projet Mr Python, ce dernier est le nom d'un environnement de développement que nous allons tenter d'améliorer lors de ce projet. Afin d'avoir une ligne directrice pour le projet, divers objectifs nous ont été présentés :

- afin de prendre en main le code de l'application, il nous a été demandé de corriger quelques bugs mineurs et d'ajouter des améliorations simples.
- Dans un second temps nous avons travaillé sur la décentralisation de l'évaluation du code écrit par l'utilisateur tout en conservant l'ensemble des fonctionnalités déjà présentes.
- Ajouter la possibilité à l'utilisateur d'interrompre son programme lorsqu'il considère que l'exécution prend trop de temps.

# Chapitre 1 : Mr Python

## I. Présentation de Mr Python et de son interface

Mr Python est un environnement de développement destiné aux élèves de première année de licence d'informatique de l'UPMC. Il est issu de l'IDE « idle », c'est Frederic Peschanski qui l'a adapté afin de le rendre plus facile à utiliser pour des débutants en programmation. Si l'on s'en tient à la version actuelle, son principal intérêt réside dans sa simplicité, en effet son utilisation est intuitive et permet à un étudiant de première année une prise en main rapide. Il existe déjà la possibilité d'interpréter le code de deux manières, l'un en mode « student » et l'autre en mode « full ». Aujourd'hui il y a assez peu de différences entre les modes, mais les souhaits de Mr Peschanski pour le mode « student » sont multiples :

- des messages d'erreurs plus précis
- des « warning » lorsque l'étudiant ne respecte pas les règles du PEP8[cite] grâce à une analyse statique
- autoriser seulement un sous ensemble de la syntaxe Python (en interdisant d'avoir le code en dehors d'une fonction par exemple)

A côté de ça, le mode « full » considère que l'utilisateur a une connaissance approfondi du langage, c'est pourquoi il aura accès à l'ensemble de la syntaxe de python et aura des messages d'erreurs classique de l'interpréter Python.



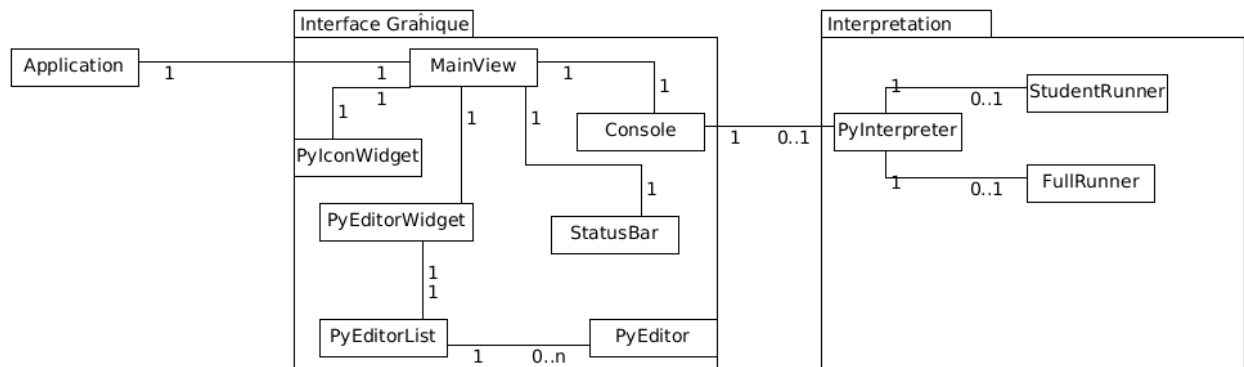
Voici un aperçu de l'interface de Mr Python, en haut nous pouvons observer les icônes qui permettent une utilisation intuitive de l'outil. Avec de gauche à droite :

- Création d'un nouveau fichier vierge
- l'ouverture d'un fichier déjà présent dans vos documents
- la sauvegarde d'un fichier
- le changement de mode d'interprétation (« student » vers « full » ou inversement)
- l'interprétation du fichier courant dans le mode sectionné grâce au bouton précédent

La partie central contiendra le texte du fichier que vous avez ouvert, à noter que le numéro des lignes apparaîtront également sur la partie gauche de l'interface lorsqu'on ouvrira un fichier. Quand à la partie inférieure de l'interface, on peut distinguer deux éléments différents. La barre jaune situé tout en bas de l'interface nous permet de passer des commandes python pouvant être directement interprété grâce à au bouton eval. La « console » permettra d'afficher les retours de l'interpréter, que ce soit les messages d'erreurs ou tout simplement les affichages demander par l'utilisateur.

## II. Structure de MrPython

Avant de vous décrire structure globale de Mr Python, nous vous proposons un diagramme de classe simplifié permettant de comprendre l'organisation.



Le fichier Application.py lance l'application, en effet c'est à partir de cette classe qu'est déclenché l'initialisation de l'ensemble de l'« ide »

Application.py : Un grand nombre d'éléments sont initialisés dans ce fichier, mais l'élément le plus important est l'initialisation de la classe MainView. Plusieurs autres fonctions importantes sont également implémentées dans Application.py, tel que le changement de mode d'interprétation, l'ouverture d'un fichier, mais également la sauvegarde.

MainView.py : Lors de sa phase d'initialisation, cette classe permet le lancement de tous les morceaux de la « vue », avec par exemple la barre de tâche, l'éditeur de texte, la console et la barre de statut. À noter également que c'est dans cette classe que la fenêtre initiale de Tkinter est lancée, en effet tous les éléments graphiques créés après seront rattachés à cet élément.

StatusBar.py, PyIconWidget.py, PyEditorWidget.py, Console.py : chacune de ces classes va créer un élément de l'interface graphique. Aux seins de leur fichier respectif, nous pouvons trouver l'implémentation de fonctions réalisant des modifications de l'interface graphique lors de l'exécution de l'application. Chaque instance de la classe PyEditor représente un onglet d'éditeur de texte, et permet de réaliser toutes les opérations nécessaires pour le bon fonctionnement de l'application.

PyEditorList va s'occuper de l'affichage graphique des PyEditor grâce à l'utilisation d'un module de Tkinter appelé Notebook. Cet élément est rattaché à une fenêtre parente et il réalisera ses affichages dans l'espace qui lui a été alloué. Le grand avantage du Notebook est de pouvoir ajouter et supprimer des éléments grâce à des fonctions simples d'utilisation. De plus lorsque deux PyEditor sont ouverts simultanément ils sont présents à l'écran sous forme d'onglet, ce qui rend l'expérience utilisateur plus agréable. Lorsque qu'un utilisateur souhaite ouvrir un nouveau fichier, une instance de PyEditor est créée, cette dernière est ensuite ajoutée au Notebook préalablement créé dans par la classe PyEditorList.

Partie interprétation : FullRunner.py, StudentRunner.py, PyInterpreter.py : C'est dans ces fichiers qu'est réalisé l'interprétation du programme. Une instance de PyInterpreter est créée dans la classe Console lorsque l'utilisateur souhaite exécuter son programme. Ensuite la classe PyInterpreter regarde le mode (Étudiant/complet) d'interprétation demandé par l'utilisateur afin d'appliquer le bon cheminement au code passé en entrée. Dans le cas d'un mode étudiant une instance de la classe StudentRunner est créée, quand au mode « full » on crée une instance de FullRunner. Il est également intéressant de noter que lorsque l'utilisateur entre une commande dans la console, elle est évaluée avec l'interpréteur déjà créé préalablement. Ce qui implique que si des variables ont été créées préalablement il est toujours possible de les afficher en passant la commande adéquate dans la fenêtre d'évaluation.

## II. Un exemple d'utilisation

### II.1 Visuel de l'exemple

Les images suivantes vont mettre en évidence l'utilisation de Mr Python tel que nous l'avons découvert



Ci-dessus, voici l'interface tel que nous vous l'avons présenté précédemment, nous allons maintenant cliquer sur le bouton permettant de créer un nouveau fichier. A noter que lorsqu'on passe notre souris sur le bouton, l'utilité du bouton et le raccourci permettant de l'utiliser apparaisse.



The screenshot shows the MrPython IDE interface. The title bar reads "exemple.py (/users/nfs/Etu6/3605386/Documents/exemple.py) - MrPython". The toolbar includes icons for editing, saving, opening, and running. The "Interpréter Ctrl-R" button is highlighted. The editor window shows the following code in "exemple.py":

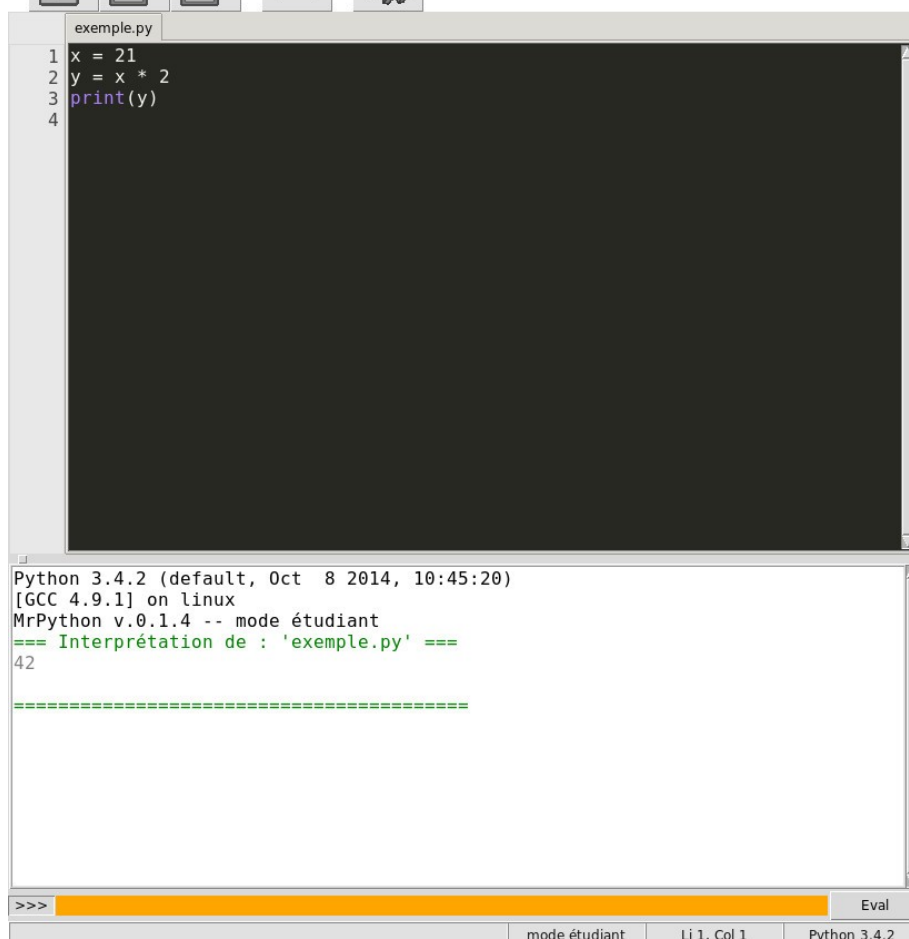
```
1 x = 21
2 y = x * 2
3 print(y)
4
```

The console window at the bottom displays the following text:

```
Python 3.4.2 (default, Oct 8 2014, 10:45:20)
[GCC 4.9.1] on linux
MrPython v.0.1.4 -- mode étudiant
```

The status bar at the bottom indicates "mode étudiant", "Li 1, Col 1", and "Python 3.4.2".

Dans un premier temps nous avons écrit un programme dans l'éditeur de texte. Après ça nous l'avons sauvegarder sous le nom de exemple.py. Nous sommes maintenant prêt à lancer l'interprétation en mode étudiant.



This screenshot shows the same MrPython IDE after the code has been executed. The code in the editor remains the same:

```
1 x = 21
2 y = x * 2
3 print(y)
4
```

The console window now shows the output of the program:

```
Python 3.4.2 (default, Oct 8 2014, 10:45:20)
[GCC 4.9.1] on linux
MrPython v.0.1.4 -- mode étudiant
=== Interprétation de : 'exemple.py' ===
42
=====
```

The status bar at the bottom remains the same, showing "mode étudiant", "Li 1, Col 1", and "Python 3.4.2".

L'interprétation a réussi car les messages affichés dans la fenêtre de sortie sont en vert, ce qui correspond au retour standard de l'interpréter Python. L'affichage de y donne donc 42, ce qui correspond à ce qu'on attendait.

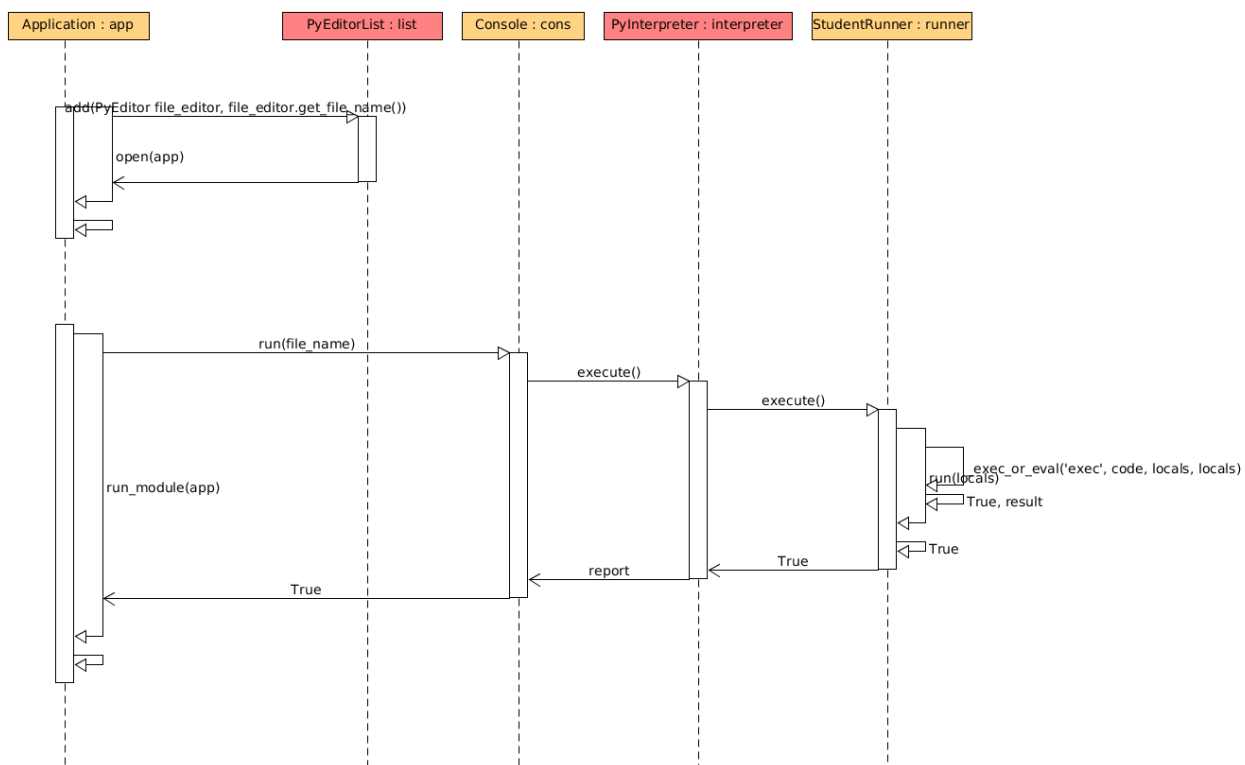


## II.2 Diagramme de séquence de l'exemple

La séquence présentée en dessous correspond à l'étape que nous avons réalisée dans l'interface graphique précédemment. Par souci de lisibilité les étapes liant le bouton jusqu'au lancement de la fonction principale située dans la classe Application seront laissés de côté.

Lorsque l'utilisateur appuie sur le bouton créant un nouvel éditeur, la fonction `open` de la classe `Application` est appelée, ensuite, celle-ci appellera la fonction `add` qui permet d'ajouter un nouvel éditeur à `PyEditorList`.

Ensuite vient l'interprétation du programme réalisé par l'utilisateur, ce dernier clique sur le bouton lançant l'interprétation, la fonction `run_module` est lancée. Ensuite toutes les fonctions permettant l'interprétation du code et la génération d'un rapport sont exécutées successivement. Dans notre cas le `PyInterpreter` détecte que nous sommes en mode « student », c'est pourquoi la fonction d'exécution de `StudentRunner` est appelée. Les éventuelles messages d'erreurs et le retour de l'interpréteur sont ensuite récupérés par la console qui s'occupera de les afficher à l'utilisateur.



## II. Correction de divers bugs

//Il manque encore les images des corrections des bugs

Dans un premier temps nous avons souhaité ajouter un bouton permettant de fermer l'éditeur courant. La commande CTRL-w permettait déjà de réaliser cette action, donc notre travail consiste à mettre en relation le bouton que l'on va créer et cette fonction déjà existante. Les principales modifications ont été réalisées dans le fichier PyEditorList.py, en effet, c'est ici que nous créons le bouton lorsque le nombre d'éditeur passe de zéro à un, mais c'est également à cet endroit qu'il est détruit lorsque qu'il n'y a plus d'éditeur ouvert. Il a également été nécessaire de modifier un attribut de la classe PyEditorWidget afin de pouvoir accéder à la bonne zone lorsque l'on souhaite créer le bouton. En effet la zone choisie ne se situait pas avec les autres boutons déjà présents, il n'était donc pas possible de le créer en même temps que ces prédécesseurs.

Le second bug qui a été corrigé est que la zone contenant les numéros de lignes ne s'étendait pas jusqu'en bas de la page lorsque l'on réduisait la console. Il se trouve que par défaut l'élément ne prenait pas toute la place lorsque l'on le redimensionnait, il fallait donc utiliser la commande rowconfigure afin de lui faire prendre la place totale. La correction de ce type de bug peut s'avérer extrêmement délicate, une personne connaissant parfaitement les spécificités de Tkinter corrigera le bug très rapidement, alors que quelqu'un n'ayant pas l'habitude pourra passer des heures avant de trouver la solution.

La correction du premier bug nous a permis de prendre en main l'architecture de Mr Python, mais elle a aussi mis en lumière des faiblesses dans la conception du logiciel. Il se trouve que le logiciel semble manquer d'une certaine cohérence globale, sans doute liée au fait que de nombreuses fonctionnalités ont été ajoutées au cours du temps à « idle ». L'un des objectifs de moyen-long terme de Mr Python est de rendre son architecture plus élégante et plus facile à comprendre pour une personne découvrant le code.

//Il manque un petit texte faisant le lien entre les deux parties

# Chapitre 2 : Serveur d'exécution

Dans notre version de MrPython, nous souhaitons séparer l'affichage graphique et l'interprétation du code de sorte à ce que ces 2 étapes aient lieu dans des processus différents.

## I. Pourquoi décentraliser?

Dans la version initiale de MrPython, l'exécution d'un programme a lieu dans le même processus qui gère l'application MrPython. Cela fait que si l'on exécute un programme sur l'interprète de MrPython, on doit attendre que l'exécution du programme soit finie pour utiliser l'application. Dans le cas où l'exécution du processus ne se termine pas, lors d'une boucle infinie par exemple, l'application entière est bloquée et on est obligé d'interrompre l'application sur le terminal.

C'est pour résoudre ce problème que nous avons dû séparer la gestion de l'application et de l'affichage graphique de l'interprétation du code python. L'exécution d'un code se fera dans un processus séparé et la gestion de l'application ne sera donc pas bloquée. De plus, on pourra interrompre à tout moment une exécution qui durerait trop longtemps.

## II. Mise en place

### II.1 Protocole Client-Serveur

Pour mettre en place ce système, nous avons utilisé un protocole Client-Serveur. Le client MrPython envoie des requêtes et recevra des réponses du serveur RunServer. Les messages seront envoyés dans le format Json. Ils seront de cette forme:

```
{
  "session_id" : uuid # identifiant de la session (session = instance de l'interprète)
  , "msg_id" : uuid # identifiant unique du message
  , "msg_type" : str # type de message (évaluation, etc.)
  , "protocol_version": str # exemple : "0.1"
  , "content" : json, # contenu du message
}
```

On définit également une structure de données data qui encode les valeurs retournées par le RunServer:

```
{ "data": [ { "mime" : str # type mime du résultat (ex. text, text/html,
                                     # image/png, etc.)
              "value" : str # encodage des données
            } ... ]
}
```

Puis nous établissons le protocole de chaque type de message:

## Évaluation d'une expression

### 1. Émission par MrPython

```
{...
  "msg_type": "eval"
  "content": {
    "expr": str # expression à évaluer
    "mode": "student" | "full" #mode d'évaluation
  }
}
```

### 2. Retour par RunServer

#### •variante 1 : succès

```
{...
  "msg_type": "eval_success"
  "content": {
    "data": data # valeur(s) retournées (cf. structure "data")
    "stdout" : str # sorties standard
    "stderr" : str # sorties erreur
    "report" : [ ... cf. "error" (mais que des warnings) ]
  }
}
```

#### •variante 2 : erreur

```
...
{...
  "msg_type": "eval_error"
  "content": {
    "report": [
      { "error_type": "compilation" | "student" | "exécution"
        "infos": { "student_error_type": str
          "severity" : "error" | "warning"
          "description": json # en fonction du type d'erreur
            } ... ]
    }
  }
}
```

Pour l'évaluation d'une expression, MrPython envoie une expression à évaluer au RunServer. RunServer retourne la valeur de l'expression, les valeurs des différents affichages et un rapport d'erreur. En cas d'erreur il ne renvoie pas de valeur pour l'expression.

Le rapport d'erreur est une liste d'erreurs. Les erreurs contiennent:

- Une chaîne de caractères nommé « error\_type » indiquant le type: compilation pour les erreurs de compilation, student pour les erreurs de convention étudiants, exécution pour les erreurs d'exécutions.
- Un objet json nommé « infos » contenant des détails sur l'erreur ainsi que son niveau de sévérité (error ou warning).

## Exécution d'un programme

### 1. Émission par MrPython

```
{...
  "msg_type": "exec"
  "content": {
    "source": str # code source à exécuter
    "mode": "student" | "full"
    "filename": str # nom du fichier
  }
}
```

### 2. Retour par RunServer

- variante 1 : succès

```
{...
  "msg_type": "exec_success"
  "content": {
    "stdout" : str # sorties standard
    "stderr" : str # sorties erreur
    "report" : [ ... cf. "error" (mais que des warnings) ]
  }
}
```

- variante 2 : erreur

```
...  
{...  
  "msg_type": "exec_error"  
  "content": {  
    "report": [  
      { "error_type": "exception" | "student"  
        "infos": { "student_error_type": str  
                    "severity" : "error" | "warning"  
                    "description": json # en fonction du type d'erreur  
                      } ... ]  
    }  
  }  
}
```

Pour l'exécution d'un programme, MrPython envoie la chaîne contenant le code source à exécuter au RunServer. RunServer retourne les valeurs des différents affichages et un rapport d'erreur et indique si l'exécution a réussi ou non.

## Interruption d'un programme

### 1. Émission par MrPython

```
{...  
  "msg_type": "interrupt"  
}
```

### 2. Retour par RunServer

- variante 1 : succès

```
{...  
  "msg_type": "interrupt_success"  
}
```

- variante 2 : erreur

```
...  
{...  
  "msg_type": "interrupt_error"  
}  
...
```

MrPython envoie un message d'interruption au RunServer. Si il n'y a pas de processus en cours d'exécution, celui ci lui indique que l'interruption à échoué.

## II.2 Implémentation

### II.2.a Schéma Général

Pour l'implémentation, nous avons créé un serveur qui fonctionne ainsi:

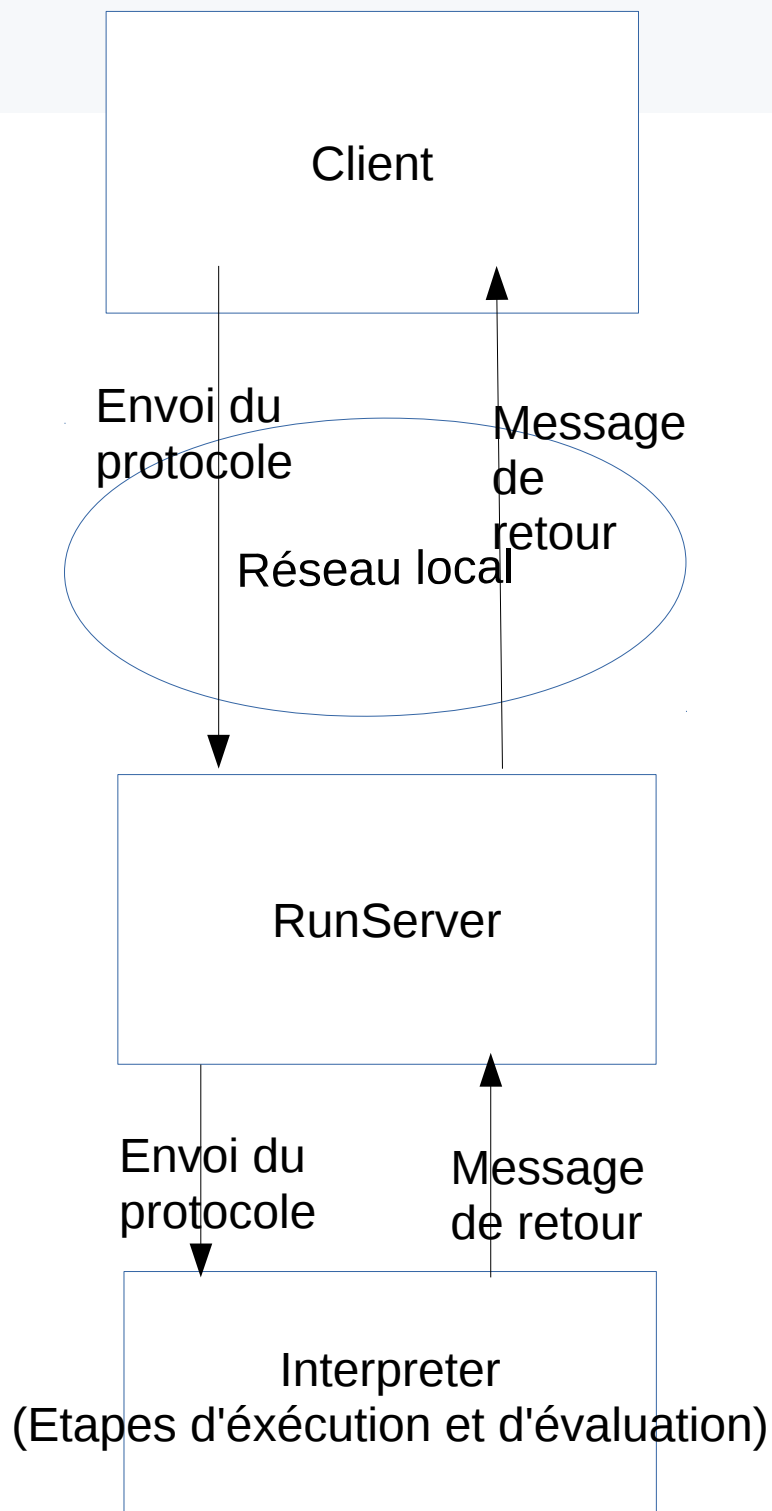


Schéma de communication entre le client et le RunServer.

Nous avons d'abord créé un client ad hoc qui envoie des requêtes d'exécution, d'évaluation et d'interruption.

Le serveur a au démarrage une instance vivante d'un Interprète qui n'exécute aucun code. Selon le type du message, le serveur exécute les actions suivantes :

- Si il s'agit d'un message d'exécution, le serveur arrête l'exécution de l'interprète en cours et crée un nouvel interprète en lui envoyant le protocole à évaluer. Il récupère ensuite le résultat calculé par l'interprète et le renvoie au client.
- S'il s'agit d'un message d'évaluation, le serveur envoie le protocole à l'interprète sans changer d'interprète puis récupère le résultat pour le renvoyer au client.
- S'il s'agit d'un message d'interruption, le serveur vérifie si l'interprète a un programme en cours d'exécution. Si oui, il arrête l'exécution en cours et envoie un message de type « interrupt\_success » au client. Si non, il envoie un message de type « interrupt\_fail ».

## II.2.b Interprète

Le fonctionnement de l'interprète est le suivant:

- Si il a reçu une requête d'évaluation ou d'exécution, il crée un processus d'exécution nommé ExecProcess qui va calculer le résultat de cette requête.
- Sinon, il attend de recevoir une requête.

Il y a 2 interprètes, un StudentInterpreter et un FullInterpreter selon le mode d'exécution donné par le client. Au départ, le serveur lance une instance de StudentInterpreter.



ExecProcess calcule le résultat de la requête en suivant un certain schéma :

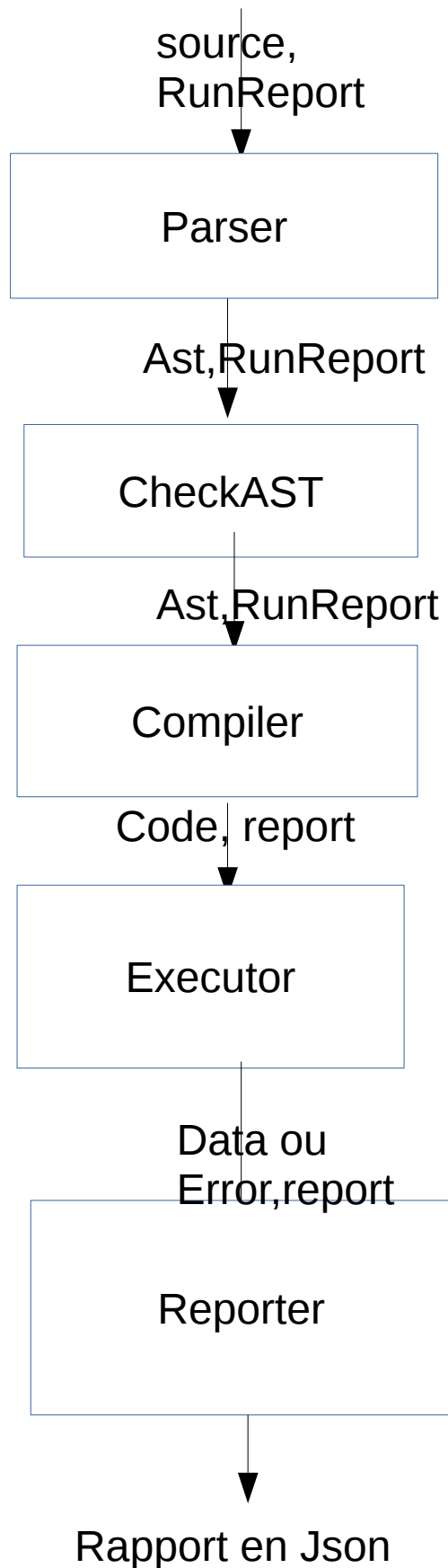


Schéma d'une exécution avec StudentExecProcess

A son initialisation, la classe StudentInterpreter possède des instances des classes Parser, CheckAST, Compiler, Executor et Reporter(nous définirons leur rôle par la suite). La classe FullInterpreter possède des instances des mêmes classes exceptées celle de la classe CheckAST.

Lors d'une exécution, ExecProcess va créer une instance de RunReport, classe de rapport d'erreurs, vide initialement.

Il va ensuite suivre une chaîne :

- Il envoie à une instance de la classe Parser le chaîne de caractère du code source à exécuter et le rapport d'erreurs. S'il n'y a pas d'erreurs, le Parser retourne l'AST du code source et un rapport d'erreur puis le transmet soit ,au CheckAst si l'exécution se fait en mode Student soit, au Compiler. Sinon il envoie le rapport d'erreur modifié au Reporter.
- La classe CheckAst n'est utilisé que par le StudentInterpreter. Elle vérifie sur un AST que les règles de syntaxe pour les étudiants ont été respectées. Si oui elle transmet l'AST au Compiler avec le rapport d'erreurs. Sinon elle transmet le rapport d'erreur modifié au reporter.
- La classe Compiler récupère un AST et un rapport d'erreurs puis exécute la compilation de l'AST. Si il n'y a pas d'erreur, il renvoie un objet code du code source et un rapport d'erreurs. Sinon il envoie le rapport d'erreurs modifié au Reporter.
- La classe Executor récupère un rapport d'erreur et un code pour l'exécuter. Il envoie ensuite un rapport d'erreurs au Reporter.
- Le reporter récupère un RunReport et le transforme en une liste de dictionnaires d'erreurs afin que le rapport d'erreurs puisse être sérialisé en Json pour être retourné au client.

Lors d'une évaluation, ExecProcess ne passe par les cases Parser et CheckAST :

- Le Compiler reçoit la chaîne de caractère de l'expression à évaluer ainsi qu'un rapport d'erreurs et retourne un code et un rapport d'erreurs s'il n'y a pas d'erreurs. Sinon, il envoie un rapport d'erreurs modifié au Reporter.
- L'Executor reçoit un code et un rapport d'erreur et renvoie une donnée contenant la valeur de l'expression évaluée ainsi qu'un rapport d'erreurs s'il n'y a pas d'erreurs. Sinon, il envoie un rapport d'erreurs modifié au Reporter.

## III.Points Techniques et Difficultés

### III.1 Gestion du port

Pour que le client et le serveur puissent communiquer ensemble, ils doivent se connecter sur un même port. On peut choisir de mettre un port statique qui sera utilisé par le client et le serveur mais cela cause un problème. En effet, après une première utilisation, il est possible que le port ne soit plus disponible. Dans ce cas, on est obligé d'attendre que le port redevienne disponible pour pouvoir utiliser l'application. Pour pallier à ce problème, nous utilisons dans notre application, un système de port dynamique. Voici son fonctionnement :

- Le serveur récupère le premier port disponible avec la méthode `"serverSocket.bind((host, 0))"`
- Il garde ensuite le numéro du port : `"port = serverSocket.getsockname()[1]"`
- Il ouvre le fichier de config "config.txt"(on considère que celui-ci est situé dans le même répertoire que celui du client et du serveur) et y écrit le numéro de port :
  - `config_file=open("config.txt","w")`
  - `config_file.write(str(port))`
  - `config_file.flush()`
- Ensuite, pour se connecter au serveur le client récupère le numéro de port dans le fichier de config:

- `mon_fichier_config = open("config.txt", "r")`
- `PORT = int(mon_fichier_config.read())`

## III.2 Communication entre le serveur et l'interprète

La fonction principale de RunServer est une boucle d'attente nommé "ServerLoop" qui attend de recevoir sur une socket une requête du client. Lorsqu'il reçoit un message d'exécution, le serveur termine le processus d'exécution de l'interprète en cours et crée une nouvelle instance d'interprète dans lequel il passe la requête du client en paramètre. Lorsque que RunServer reçoit un message d'évaluation, il ne change pas l'instance de l'interprète mais doit lui communiquer le message. Pour cela, on utilise un pipe de la bibliothèque multi-processing, On crée notre pipe ainsi :  
`server_con, interpreter_conn = Pipe()`.

"Server\_con" est la connexion du serveur et "Interpreter\_conn" est la connexion de l'interprète. Celle-ci est passé en paramètre de l'interprète à sa création. Le serveur peut donc envoyer des messages sur sa connexion que l'interprète recevra sur la sienne et vice-versa.

## III.3 Sécurité du serveur

## III.4 Gestion des variables locales

## III.5 Sérialisation en Json

## IV. Dans le Futur

### IV.1 Gestion des erreurs étudiants

### IV.2 Entrée sur console

### IV.3 Affichage graphique

Conclusion: