# Block2Py : retour sur les choix didactiques du projet

Christophe Declercq, LIM, IREMI

Septembre 2024

## Introduction

L'éditeur Block2Py a été proposé en 2020 [1] pour faciliter la transition d'un langage de programmation par blocs à un langage textuel. La version impérative a depuis été améliorée par l'ajout d'une interface d'exécution et est disponible à l'adresse :

`https://iremi974.gitlab.io/block2py/imperatif.html`

Plusieurs expérimentations ont été menées avec cet environnement dont les dernières dans le cadre du mémoire de master MEEF de Velaven Vingadassalom à l'INSPE de l'académie de La Réunion [3].

En 2023, nous avons aussi proposé [2] une version destinée à la programmation fonctionnelle, disponible à l'adresse :

`https://iremi974.gitlab.io/block2py/fonctionnel.html`

Dans ce rapport, nous explicitons les choix didactiques ayant conduit à la conception de ces environnements en précisant en particulier le choix de sous-ensembles du langage Python et les choix instrumentaux ayant conduit à la définition des interfaces.

## Le langage impératif élémentaire

Choisir un sous-ensemble du langage Python pour l'enseigner est une démarche de transposition didactique qui doit être menée avec précaution par l'enseignant, sachant qu'aucune expertise du langage par les élèves n'est visée, mais que le langage doit être le support d'apprentissages en programmation. Les notions de programmation au programme de la classe de seconde contiennent des notions déjà abordées au collège en programmation par blocs : affectations, variables, séquences, instructions conditionnelles, boucles bornées et non bornées et une nouvelle notion : définitions et appels de fonctions.

C'est pour travailler spécifiquement la transition, que nous avons choisi de n'aborder dans un premier temps que les notions présentes à la fois au programme du collège et de la seconde et donc d'exclure la notion de fonction du premier langage impératif élémentaire proposé. Ce choix a aussi l'avantage de simplifier le modèle de machine sous-jacent.

## La distinction instruction / expression

La confusion entre une instruction - dont l'exécution fait quelque chose - et une expression - dont l'évaluation donne une valeur, est une cause fréquente d'erreurs de syntaxe chez les programmeurs débutants. Le langage Python n'aide pas les élèves dans cette distinction, certaines instructions y étant considérées comme des expressions dont l'évaluation vaut `None`. Par exemple, Python accepte la phrase `x = print('Bonjour')` que nous considérons comme une erreur de débutant.

Cette distinction a déjà pu être construite par les élèves en programmation par blocs : les instructions sont les blocs qui peuvent s'assembler séquentiellement par des connecteurs en haut et en bas ; les expressions sont les blocs qui disposent d'un connecteur à gauche renvoyant sa valeur. Nous avons fait le choix de bien distinguer instructions et expressions en proposant la syntaxe abstraite suivante (extrait):

```
<instruction> ::= print(<expression>) | <variable> = <expression> |
                  if <expression> : <instructions> |
                  if <expression> : <instructions> else : <instructions> |
                  while <expression> : <instructions> |
                  for <variable> in range (<expression>) : <instructions>

<expression> ::= <variable> | <constante> | <expression> + <expression> |
                 <expression> * <expression> | ...
```

Le `print` est, dans cette syntaxe, une instruction, dont l'exécution aura pour effet d'afficher un message à l'écran. Cette syntaxe permet de limiter aux expressions ce que l'on peut écrire en partie droite d'une affectation. Cela évite ainsi des écritures autorisées en Python comme `x = y = 42`, que l'on préfère proscrire pour des débutants. Il est inutile de décrire la valeur `None` pour expliquer l'évaluation d'une instruction, puisque dans notre modèle, une instruction n'est pas évaluée mais exécutée.

La manipulation avec Block2Py de blocs, correspondant à des expressions ou à des instructions, permet à l'élève de construire des programmes Python corrects par construction au niveau syntaxique. Des exemples de progression pour le début de la classe de seconde ont été proposées en utilisant Block2Py [3].

## Les variables et le modèle de mémoire

La complexité du modèle de mémoire de Python peut aussi être masquée aux programmeurs débutants. Dans le cadre de la programmation impérative ne

manipulant que des types élémentaires, sans fonctions ni procédures, on peut simplement considérer la mémoire comme une fonction qui à un nom associe une valeur. L'affectation consiste alors à évaluer l'expression en partie droite, puis à mémoriser la valeur obtenue sous le nom inscrit en partie gauche de l'affectation.

Dans l'interface de Block2Py, on contraint de plus le programmeur à indiquer au préalable les variables dont il a besoin en précisant leur type. C'est bien sûr une contrainte plus forte pour le programmeur, que ce que Python exige. C'est la condition pour qu'un contrôle statique des types puisse être effectué. Les expérimentations ont montré [3] que cette politique évite de nombreuses erreurs de types aux programmeurs débutants : en particulier dès qu'une saisie est effectuée, l'oubli d'une conversion entraîne des erreurs lors de calculs arithmétiques opérés sur des chaînes de caractères.

Pour que l'élève construise un modèle mental cohérent avec ce modèle de mémoire, il peut être utile que l'environnement d'apprentissage procure un retour instrumental adapté, montrant les variables et l'évolution de leurs valeurs.

Un retour instantané, doit montrer les valeurs de chaque variable, tout au long d'une exécution pas à pas d'un programme.

Ce retour peut aussi être de type historique et montrer les valeurs de chaque variable aux points d'arrêt définis pour l'exécution. Ce retour prend alors la forme d'un "tableau d'exécution", avec en première ligne, la liste des variables, puis sur les lignes suivantes, leurs valeurs à différents instants successifs de l'exécution.

Ce modèle de mémoire est celui appelé "modèle à boîtes" dans le travail de classification effectué par Exibard et al. [4]. La métaphore de la boîte ayant un contenu est cependant à utiliser avec prudence, car affecter une variable `x` à la valeur de la variable `y`, n'est pas vider une boîte dans l'autre : l'affectation n'efface pas mais recopie le contenu. On préférera parler de mémoire effaçable et utiliser comme métaphore de la variable, l'ardoise effaçable.

## Ajout de fonctions et procédures au langage élémentaire

Pour prolonger la distinction entre instructions et expressions, il convient de distinguer fonctions et procédures, même si le langage Python ne fait pas cette distinction, car il propose une notion englobant les deux.

On peut appeler *procédure élémentaire*, toute fonction Python ne possédant pas de clause `return`. L'appel d'une procédure est une instruction. On peut appeler *fonction élémentaire*, toute fonction Python possédant une et une seule clause `return` positionnée à la fin du texte de la fonction. Une fonction élémentaire est dite *pure* si elle ne contient aucune instruction avant la clause `return`. L'appel d'une fonction est une expression.

Les procédures ont une utilité pour leur effet collatéral : affichage d'un dessin ou interaction avec l'utilisateur. Les fonctions servent à prévoir un calcul défini

par une expression - en mathématiques par exemple.

Le modèle de mémoire présenté précédemment reste valide pour expliquer le passage de paramètres, tant que l'on n'utilise que des types élémentaires (entier, booléen et chaînes de caractères). Le passage de paramètre peut être décrit comme une évaluation des paramètres effectifs puis une affectation aux paramètres formels des valeurs obtenues. L'évaluation de l'appel à une fonction consiste à évaluer sa clause `return`.

### Limites du modèle de mémoire

Si on introduit dans le langage, la notion de tableau de Python, ou toute autre donnée mutable, il peut être nécessaire de réviser le modèle de mémoire et de le remplacer par un modèle environnement / mémoire. L'environnement est une fonction qui à chaque nom de variable associe une adresse (ou référence). La mémoire est une fonction qui à chaque adresse (ou référence) associe une valeur.

Ce modèle est nécessaire pour décrire le phénomène d'alias avec des données mutables. Il est indispensable dès que l'on souhaite pouvoir donner un sens à l'affectation d'une variable contenant un tableau à une autre variable ou si l'on souhaite passer un tableau en paramètre d'une fonction.

On peut cependant conserver le modèle de mémoire élémentaire et étudier les tableaux en Python dans le cadre du programme de première à condition de s'interdire toute affectation globale de tableau ainsi que tout passage de tableau en paramètre. Cela n'empêche pas de décrire les algorithmes au programme (tri, recherche...) en utilisant uniquement pour les tableaux, l'affectation composante par composante. Ceci revient à considérer un tableau comme une collection indicée de variables et à les représenter comme telles.

## Le langage fonctionnel élémentaire

Dans le contexte de la programmation fonctionnelle dite "pure", toute fonction ne possède qu'une clause `return` composée d'une expression décrivant le résultat à calculer [2]. On peut donc limiter la syntaxe au domaine des expressions et supprimer toutes les instructions. On a montré que la pratique de la programmation fonctionnelle au niveau élémentaire, est un préalable utile à l'apprentissage de la récursivité.

### Le modèle de machine

On a montré [2] que la console est un instrument fondamental en programmation fonctionnelle, dans le sens où elle aide à faire comprendre ce que peut faire une « machine fonctionnelle ». Une telle machine est juste capable, si on lui a fourni le texte définissant une fonction, d'évaluer les appels à cette fonction avec les valeurs de son choix. Cette machine n'a pas de mémoire à part pour mémoriser

les fonctions. Elle ressemble plutôt à une calculatrice, capable d'évaluer toute expression pouvant comporter un ou des appels de fonctions déjà définies.

Il n'y a pas lieu de discuter du modèle de mémoire pour la programmation fonctionnelle, puisqu'on ne fait pas directement usage de la notion de mémoire. Il n'y a pas de variable dont il faudrait suivre l'évolution des valeurs.

Il n'y a que des appels de fonction. Le seul instrument pertinent en programmation fonctionnelle, qui donne un retour instrumental constructif, est la console avec la possibilité de tracer l'ensemble des appels de fonction. C'est ainsi que le programmeur peut observer la construction progressive d'un résultat.

## Conclusions et perspectives

C'est bien dans une intention didactique de séparer les difficultés d'apprentissage, que l'on a choisi de proposer deux sous-ensembles distincts de Python, pour faire d'abord un apprentissage de la programmation impérative, puis de la programmation fonctionnelle et de la récursivité.

Il reste maintenant la possibilité d'intégrer les deux approches dans un cadre qui se rapproche de ce que le langage Python permet. Nous aurons alors la possibilité et le risque de voir le programmeur mélanger les deux paradigmes. Ce mélange peut être pertinent si le programmeur utilise à bon escient chaque paradigme. C'est en tout cas un moyen de permettre au programmeur d'étudier ces paradigmes et leur complémentarité.

## Références

[1] Christophe Declercq, Florence Nény. Block2Py, un éditeur de blocs pour l'apprentissage du langage Python. Didapro 8 – DidaSTIC, Feb 2020, Lille, France. hal-02526883

[2] Christophe Declercq, Sophie Chane-Lune, Sébastien Hoarau. Expérimentation d'un environnement pour l'apprentissage de la programmation fonctionnelle et de la récursivité en terminale NSI. Adjectif : analyses et recherches sur les TICE, 2023, T3. hal-04350417

[3] Velaven VINGADASSALOM, La transition Scratch-Python en classe de seconde. Mémoire de master MEEF. INSPE de l'académie de La Réunion, juin 2024.

[4] Léo Exibard, Nadime Francis, Antoine Meyer, Marie van den Bogaard. Modèles de mémoire pour l'enseignement de la programmation. Colloque Didapro 10 sur la Didactique de l'informatique et des STIC, 2024, Louvain-La-Neuve, Belgique. pp.33-43. hal-04482121v2