

# Modèles de mémoire pour l'enseignement de la programmation en Python

Sébastien Hoarau, Christophe Declercq

9 octobre 2024



# Plan de la présentation

- Cadre et objectifs de la réflexion
- Un premier modèle pour les débutants
- Un second modèle adapté du premier
- Un troisième modèle plus réaliste
- Conclusions et perspectives

# Cadre de la réflexion

## Cadre théorique

- L'informatique repose sur 4 piliers : des algorithmes, des langages, des machines et des informations [G. Dowek].
- On s'intéresse à l'adéquation langage  $\leftrightarrow$  machine et plus précisément à la question : quelle image de la machine pour les élèves, selon le niveau de langage à apprendre ?
- Voir la notion de “machines notionnelles” [Du Boulay]

## Contextes

- Enseignement de Python en seconde, classe de mathématiques et de SNT
- Enseignement de Python en première NSI
- Enseignement de Python en terminale NSI ou en premier cycle à l'université

# Objectifs

## Identifier des couplages cohérents

- un sous-ensemble du langage Python
- un modèle de machine
- la “sémantique” (sens) d’une construction du langage doit pouvoir être expliquée sur la machine

## Proposer des environnements d’apprentissage cohérents

- Des outils pour l’enseignant : représentations de la machine (partie de) au tableau
- Des instruments pour l’élève : IDE (environnement de programmation)

# Une première proposition pour les programmeurs débutants

## Un sous-ensemble de Python très réduit

- **Memento 2nde**
- expressions sur types simples (int, str, float, bool)
- affectations et instructions composées (séquence, if, while, for)
- fonctions (définition et appel)

## Une machine à mémoire

- la **mémoire** associe à des noms, des valeurs
- le sens d'une expression, c'est son **évaluation**, connaissant l'état de la mémoire
- le sens d'une instruction, c'est comment son **exécution** modifie la mémoire
- le sens d'une **affectation**, c'est évaluer l'expression en partie droite, puis associer cette valeur au nom figurant en partie gauche
- le sens d'un **appel de fonction**, c'est déléguer à une autre machine, l'exécution du code avec pour mémoire l'affectation des paramètres aux valeurs données à l'appel

# Représentation des variables

- une variable, c'est une **case mémoire** qui a un nom
- métaphore de la variable : l'ardoise effaçable
- l'ensemble des variables représente l'**état** de la machine à un instant donné

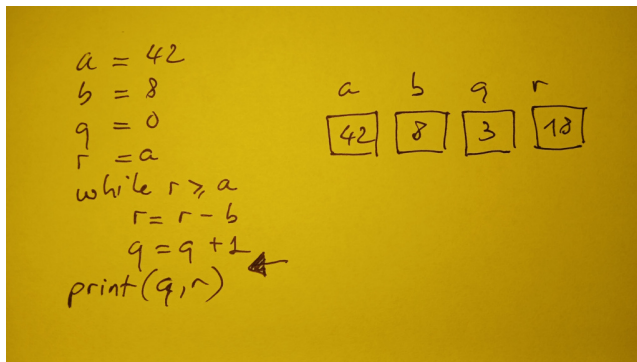
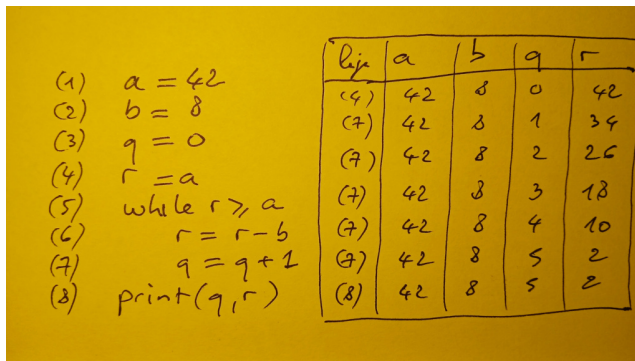


Figure 1: Etat de la mémoire

# Représentation de l'historique

- tableau d'exécution : une colonne par variable
- observation des valeurs des variables à certains moments particuliers de l'exécution : points d'arrêts
- le tableau représente l'ensemble des états observés de la machine au cours de l'exécution



Handwritten code and execution table on a yellow background.

Code:

```
(1) a = 42
(2) b = 8
(3) q = 0
(4) r = a
(5) while r >= a
(6)     r = r - b
(7)     q = q + 1
(8) print(q, r)
```

Table:

| lignes | a  | b | q | r  |
|--------|----|---|---|----|
| (4)    | 42 | 8 | 0 | 42 |
| (7)    | 42 | 8 | 1 | 34 |
| (7)    | 42 | 8 | 2 | 26 |
| (7)    | 42 | 8 | 3 | 18 |
| (7)    | 42 | 8 | 4 | 10 |
| (7)    | 42 | 8 | 5 | 2  |
| (8)    | 42 | 8 | 5 | 2  |

Figure 2: Historique des états mémoire

# Des outils et instruments pour programmeurs débutants

Mu 1.0.3 - quotient.py

Mode Nouveau Charger Enregistrer Arrêter Continuer Sauter Entrer Sortir Zoomer Dé-zoomer Thème

quotient.py ✕

```
1 a = 42
2 b = 8
3 q = 0
4 r = a
5 while r >= b:
6     r = r - b
7     q = q + 1
8 print(q, r)
```

Inspecteur pour le débogage

| Nom     | Valeur                          |
|---------|---------------------------------|
| __fi... | '/home/declercq/mu_code/quot... |
| __na... | '__main__'                      |
| a       | 42                              |
| b       | 8                               |
| q       | 3                               |
| r       | 18                              |

En cours d'exécution: quotient.py

Debugger ⚙

# Des outils et instruments pour programmeurs débutants

Thonny - /home/declercq/IREMI/Python/euclide.py @ 8 : 12

Fichier Édition Affichage Exécuter Outils Aide

euclide.py

```
1 a = 42
2 b = 8
3 q = 0
4 r = a
5 while r >= b:
6     r = r - b
7     q = q + 1
8 print(q, r)
```

Variables

| Nom | Valeur |
|-----|--------|
| a   | 42     |
| b   | 8      |
| q   | 3      |
| r   | 18     |

Console

```
Python 3.10.12 (/usr/bin/python3)
>>> %cd /home/declercq/IREMI/Python
>>> %Debug euclide.py
5 2
>>> %Debug euclide.py
```

# Une deuxième proposition avec des structures de données

## Un sous-ensemble de Python augmenté

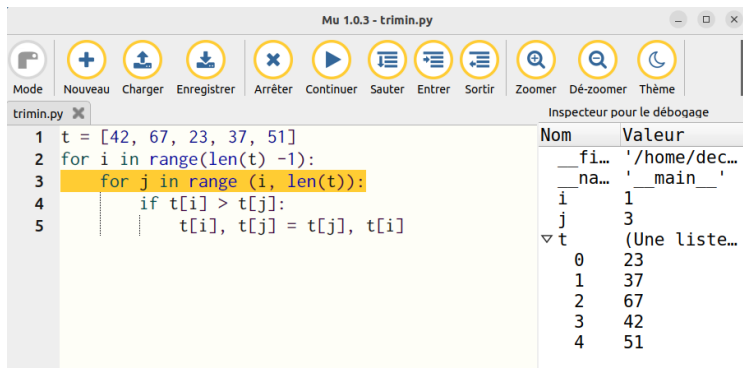
- **Memento 1ere**
- ajout des enregistrements, des tableaux, des tuples : considérés comme des collections de variables.
- Le nombre de composantes est connu de manière statique à la construction.
- instructions de création
- expression d'accès
- instructions de modification pour les tableaux et les enregistrements

## Une machine à mémoire "structurée"

- des cases contigues nommées par un indice (tableaux, tuples) ou un nom (enregistrements)
- change le format des noms de variables, la mémoire reste une association : noms -> valeurs.

# Représentation des variables

- une variable, c'est une **case mémoire** avec un nom et parfois un indice ou un nom de champ
- l'ensemble des variables représente l'**état** de la machine à un instant donné



The screenshot shows the Mu Python IDE interface. The main editor displays a Python script named `trimin.py` with the following code:

```
1 t = [42, 67, 23, 37, 51]
2 for i in range(len(t) - 1):
3     for j in range(i, len(t)):
4         if t[i] > t[j]:
5             t[i], t[j] = t[j], t[i]
```

The line `for j in range(i, len(t)):` is highlighted in yellow. To the right, the 'Inspecteur pour le débogage' (Debugger Inspector) is open, showing a table of variables and their values:

| Nom                   | Valeur                     |
|-----------------------|----------------------------|
| <code>__file__</code> | <code>'/home/dec...</code> |
| <code>__name__</code> | <code>'__main__'</code>    |
| <code>i</code>        | <code>1</code>             |
| <code>j</code>        | <code>3</code>             |
| <code>t</code>        | <code>(Une liste...</code> |
| <code>0</code>        | <code>23</code>            |
| <code>1</code>        | <code>37</code>            |
| <code>2</code>        | <code>67</code>            |
| <code>3</code>        | <code>42</code>            |
| <code>4</code>        | <code>51</code>            |

Figure 5: Visualisation des tableaux avec Mu

# Représentation des variables

- On peut aussi considérer qu'une variable a une collection de valeurs

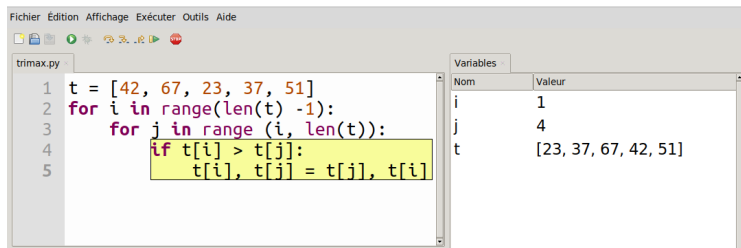


Figure 6: Visualisation des tableaux avec Thonny

# Représentation de l'historique

- on représente les structures en lignes
- le tableau représente l'ensemble des états observés de la machine

$t = [42, 67, 23, 37, 51]$   
for  $i$  in range( $\text{len}(t) - 1$ ):  
    for  $j$  in range( $i + 1, \text{len}(t)$ ):  
        if  $t[i] > t[j]$   
             $t[i], t[j] = t[j], t[i]$

| i | j | t  |    |    |    |    |
|---|---|----|----|----|----|----|
|   |   | 0  | 1  | 2  | 3  | 4  |
| 0 | 1 | 42 | 67 | 23 | 37 | 51 |
| 0 | 2 | 23 | 67 | 42 | 37 | 51 |
| 0 | 3 | 23 | 67 | 42 | 37 | 51 |
| 0 | 4 | 23 | 67 | 42 | 37 | 51 |
| 1 | 2 | 23 | 42 | 67 | 37 | 51 |
| 1 | 3 | 23 | 37 | 67 | 42 | 51 |
| 1 | 4 | 23 | 37 | 67 | 42 | 51 |
| 2 | 3 | 23 | 37 | 42 | 67 | 51 |
| 2 | 4 | 23 | 37 | 42 | 67 | 51 |
| 3 | 4 | 23 | 37 | 42 | 51 | 67 |

Figure 7: Historique

# Limites de la deuxième proposition

## Restrictions au niveau langage

- pas d'affectation globale de tableau (après la construction initiale)
- on n'autorise que l'affectation par composante du tableau : `t[4] = 12`
- pas de tableau en **paramètre** d'une fonction

## Limites du modèle mémoire

- ne peut pas expliquer les phénomènes d'aliasing (deux références sur le même tableau)
- ne peut pas expliquer pourquoi un appel de fonction peut modifier les composantes d'un tableau passé en paramètre

# Un modèle plus proche de Python

à toi Sébastien !