

Bases d'algorithmique et de programmation

Sébastien Gagné, Université d'Orléans

L1 Algo - Prog - S1

Cours 01 - Types, Variables

Plan du cours

- Types algo, correspondance python
- Autres types en python
- Le typage en python
- Variables, Affectation (algo)
- Mécanisme python de l'affectation
- Déréférencement en python
- Types itérables en python
- Types immuables ou muables en python

Types algorithmiques classiques

Voici les **types** classiques utilisés en **algorithmique** :
entier, réel, caractère, booléen, tableau.

Remarque 1

En python, un réel est typé en **float**. Dans d'autres langages on utilise **float** ou **double**, selon la précision souhaitée.

Remarque 2

En algorithmique, une chaîne de caractères peut être vue comme un tableau de caractères.

Remarque 3

En python, le type chaîne de caractères existe (**str**), mais pas le type caractère.

Correspondance algo - python

On part sur les correspondances suivantes

type algorithmique	type python
entier	int
réel	float
caractère	str
chaîne de caractères	
booléen	bool
tableau	list, tuple

Autres types en python

python propose d'autres types particuliers :

- *tableau associatif* : ***dict***
- *ensemble* : ***set***
- un type très utile : ***range*** (suite arithmétique)
- fonctions, méthodes, ...

Remarque

La fonction **type()** retourne le type de l'argument.

Typage python

python est **fortement typé**.

=> toute variable a un type définitif à sa création.

python est à **typage dynamique**.

=> le type est automatiquement attribué à la variable selon la valeur qui lui est donnée.

Variable - Affectation

En algo, une **variable** stocke une valeur (en gros, une "boîte" avec un nom et contenant une valeur).



L'**affectation** (donner une valeur à une variable) sera notée, en python

a = 1

Le type n'est pas précisé : le typage est "dynamique", et s'adapte automatiquement à la nature de la valeur.

Variable - Affectation

Selon les langages, la gestion des variables diffère.

- où est stockée la valeur en mémoire ?
- quelle est la taille de cet emplacement ?
- comment accéder à la valeur ?
- peut-on modifier la valeur ? si oui, comment ?
- que deviennent les valeurs stockées inaccessibles ?

Variable - Affectation en python

en python, une variable est une référence nommée.

Une variable est un moyen "nommé" d'accéder à un emplacement mémoire où se trouve une valeur.

Lors d'une affectation **a = 1** il y a en python :

- création du nom de variable *a* ;
- allocation d'un emplacement en mémoire pour stocker une valeur de type **int** ;
- stockage de la valeur 1 à cet emplacement ;
- création de lien entre le nom *a* et l'emplacement ;

Variable - Affectation en python

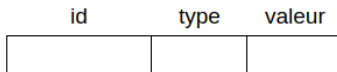
Lors d'une affectation **a = 1** il y a en python :

- création du nom de variable *a* ;

espace de noms



mémoire



Variable - Affectation en python

Lors d'une affectation **a = 1** il y a en python :

- allocation de l'espace en mémoire ;

espace de noms

a

mémoire

id	type	valeur
.....226160	int	

Variable - Affectation en python

Lors d'une affectation **a = 1** il y a en python :

- stockage de la valeur 1 dans l'espace alloué ;

espace de noms

a

mémoire

id	type	valeur
.....226160	int	1

Variable - Affectation en python

Lors d'une affectation **a = 1** il y a en python :

- création du lien entre nom et espace mémoire ;

espace de noms

a



mémoire

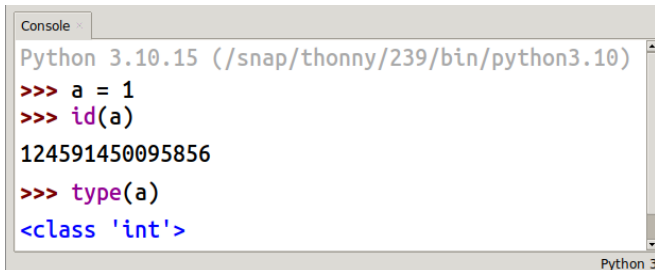
id	type	valeur
.....226160	int	1

Variable - Affectation en python

a est le lien vers la valeur **1** : **référence nommée**

L'adresse en mémoire est représentée par : **id(a)**.

Le type de la variable est obtenu par : **type(a)**



```
Console x
Python 3.10.15 (/snap/thonny/239/bin/python3.10)
>>> a = 1
>>> id(a)
124591450095856
>>> type(a)
<class 'int'>
```

The screenshot shows a Python 3.10.15 console window. The prompt is 'Python 3.10.15 (/snap/thonny/239/bin/python3.10)'. The user enters three commands: 'a = 1', 'id(a)', and 'type(a)'. The outputs are '124591450095856' and '<class 'int'>' respectively. The console window has a title bar 'Console x' and a status bar 'Python 3'.

Que se passe-t-il ici ?

```
1 a = 1
2 print("a =", a, " et id(a) =", id(a))
3
4 b = 2
5 print("b =", b, " et id(b) =", id(b))
6
7 c = 1
8 print("c =", c, " et id(c) =", id(c))
9
10 a = 3
11 print("a =", a, " et id(a) =", id(a))
12
13 c = 2
14 print("c =", c, " et id(c) =", id(c))
```

```
a = 1 et id(a) = 135659065524464
b = 2 et id(b) = 135659065524496
c = 1 et id(c) = 135659065524464
a = 3 et id(a) = 135659065524528
c = 2 et id(c) = 135659065524496
```

Que se passe-t-il ici ?

- création de a : valeur = 1, id = 135659065524464 ;

espace de noms

mémoire



Que se passe-t-il ici ?

- création de b : valeur = 2, id = 135659065524496 ;

espace de noms

mémoire

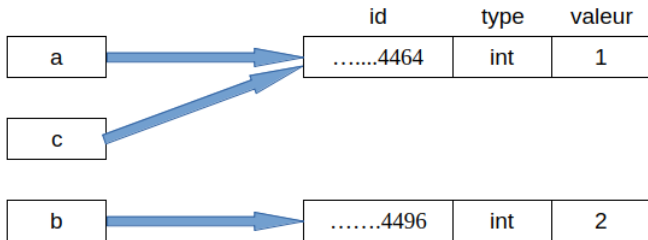


Que se passe-t-il ici ?

- création de c : valeur = 1, id = 135659065524464 ;

espace de noms

mémoire

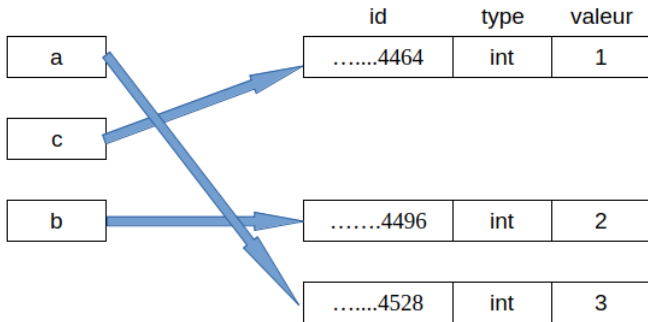


Que se passe-t-il ici ?

- affectation `a = 3` : `valeur = 3`, `id = 135659065524528` ;

espace de noms

mémoire

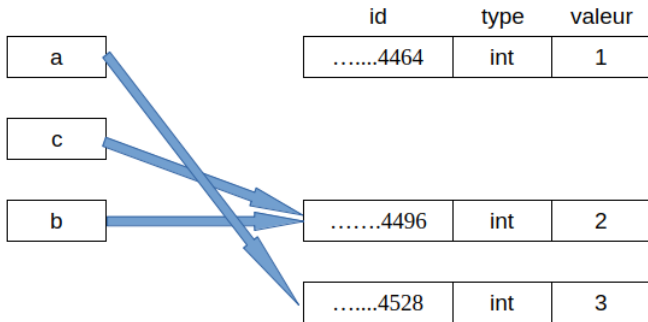


Que se passe-t-il ici ?

- affectation `c = 2` : valeur = 2, id = 135659065524496 ;

espace de noms

mémoire



Que se passe-t-il ici ?

- déréférencement d'une zone mémoire ;

espace de noms

a

c

b

mémoire

id	type	valeur
.....4464	int	1

zone mémoire déréférencée

.....4496	int	2
-----------	-----	---

.....4528	int	3
-----------	-----	---

Que se passe-t-il ici ?

1. La zone mémoire à l'adresse `id = ...4464` a été **déréférencée** (elle n'est plus accessible par le nom d'une variable).
Elle est remise à disposition par le *garbage collector* (ramasse-miettes).
2. Les variables **ne sont pas modifiées** par une nouvelle affectation.
Une **affectation** revient en fait à **recréer une nouvelle variable**.

Que se passe-t-il ici ?

```
>>> liste1 = [3, 1, 4]
>>> liste1
[3, 1, 4]
>>> id(liste1)
124591366651328
>>> liste1 = liste1 + [1, 5]
>>> liste1
[3, 1, 4, 1, 5]
>>> id(liste1)
124591366668672
```

```
>>> a = 5
>>> id(a)
124591450095984
>>> a = a + 3
>>> id(a)
124591450096080
```

Les variables *liste1* et *a* ont été recrées lors de la deuxième affectation. Ce ne sont plus les mêmes variables.

Valeurs de types classiques

- Les valeurs de type *int* : -5, -123, 0, 4, 1256, ...
- Les valeurs de type *float* : -3.4, 1.0, 2.718, 3.14159, ...
- Les valeurs de type *bool* : vrai (*True*) et faux (*False*)
- Les valeurs de type *str* : 'a', "123", 'Hello', "", ...
- Les valeurs de type *range* :
range(d, f, p) est la suite commençant à *d*, progressant de *p* en *p*, et s'arrêtant avant *f*.

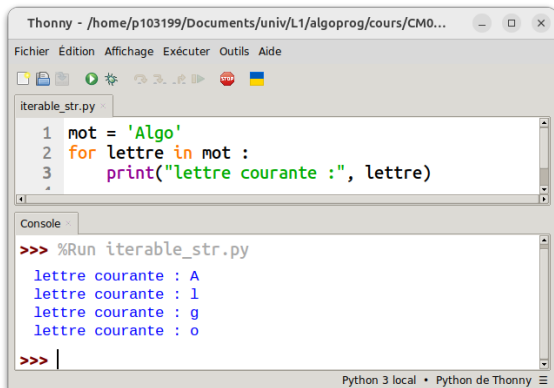
Valeurs de types classiques

- Les valeurs de type ***tuple*** représentent les tableaux de taille fixe : (2.5, -3.4, 1.2), ...
- Les valeurs de type ***list*** représentent les tableaux de taille variable : ['a', 'b', 'c'], ...
- Les valeurs de type ***set*** représentent les ensembles (pas d'ordre, ni de répétition) : {1, 2, 3}, ...
- Les valeurs de type ***dict*** représentent les dictionnaires (ou tableaux associatifs) : {'x' : 1.2, 'y' : -3.1}, ...

Types itérables en python

Un type est **itérable** si les variables de ce type peuvent être parcourues au moyen d'une boucle **for**.

Les types **str**, **tuple**, **liste**, **dict**, **range** sont itérables.



The screenshot shows the Thonny Python IDE interface. The main window displays a file named `iterable_str.py` with the following Python code:

```
1 mot = 'Algo'
2 for lettre in mot :
3     print("lettre courante :", lettre)
```

Below the code editor is a console window showing the output of the script after running it:

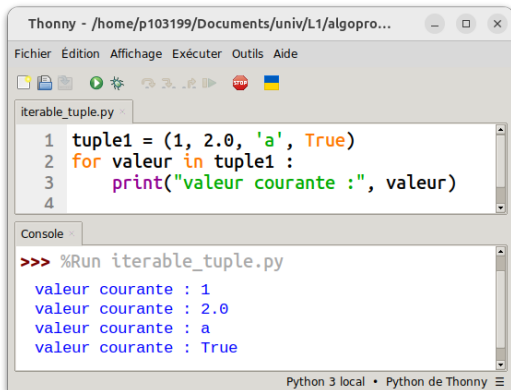
```
>>> %Run iterable_str.py
lettre courante : A
lettre courante : l
lettre courante : g
lettre courante : o
>>> |
```

The status bar at the bottom indicates "Python 3 local • Python de Thonny".

Types itérables en python

Un type est **itérable** si les variables de ce type peuvent être parcourues au moyen d'une boucle **for**.

Les types **str**, **tuple**, **liste**, **dict**, **range** sont itérables.



The screenshot shows the Thonny Python IDE interface. The title bar reads "Thonny - /home/p103199/Documents/univ/L1/algopro...". The menu bar includes "Fichier", "Édition", "Affichage", "Exécuter", "Outils", and "Aide". Below the menu is a toolbar with icons for file operations, running, and debugging. The editor window, titled "iterable_tuple.py", contains the following Python code:

```
1 tuple1 = (1, 2.0, 'a', True)
2 for valeur in tuple1 :
3     print("valeur courante :", valeur)
4
```

The console window, titled "Console", shows the output of running the script:

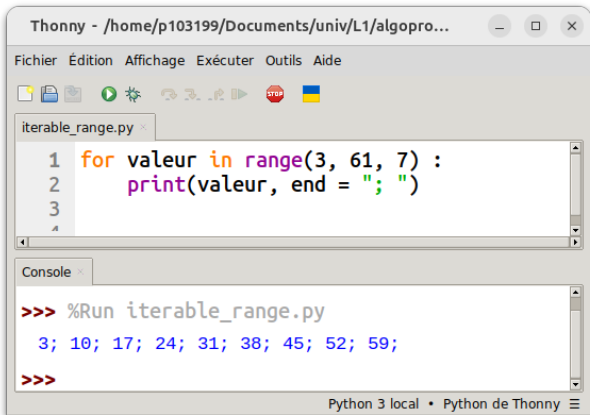
```
>>> %Run iterable_tuple.py
valeur courante : 1
valeur courante : 2.0
valeur courante : a
valeur courante : True
```

The status bar at the bottom indicates "Python 3 local" and "Python de Thonny".

Types itérables en python

Un type est **itérable** si les variables de ce type peuvent être parcourues au moyen d'une boucle **for**.

Les types **str**, **tuple**, **liste**, **dict**, **range** sont itérables.



The screenshot shows the Thonny Python IDE interface. The title bar reads "Thonny - /home/p103199/Documents/univ/L1/algopro...". The menu bar includes "Fichier", "Édition", "Affichage", "Exécuter", "Outils", and "Aide". The toolbar contains icons for file operations, running, and debugging. The editor window, titled "iterable_range.py", contains the following Python code:

```
1 for valeur in range(3, 61, 7) :  
2     print(valeur, end = "; ")  
3
```

The console window, titled "Console", shows the execution output:

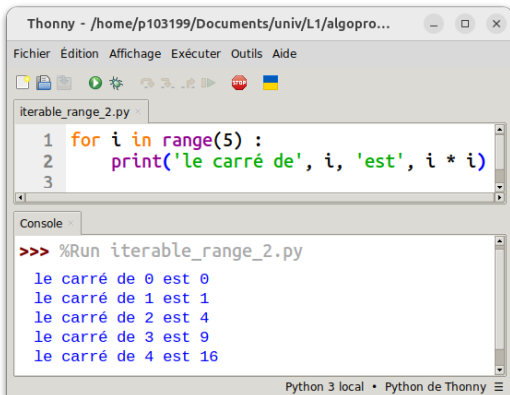
```
>>> %Run iterable_range.py  
3; 10; 17; 24; 31; 38; 45; 52; 59;  
>>>
```

The status bar at the bottom indicates "Python 3 local" and "Python de Thonny".

Types itérables en python

Un type est **itérable** si les variables de ce type peuvent être parcourues au moyen d'une boucle **for**.

Les types **str**, **tuple**, **liste**, **dict**, **range** sont itérables.



The screenshot shows the Thonny Python IDE interface. The title bar reads "Thonny - /home/p103199/Documents/univ/L1/algopro...". The menu bar includes "Fichier", "Édition", "Affichage", "Exécuter", "Outils", and "Aide". The toolbar contains icons for file operations, running, and debugging. The editor window, titled "iterable_range_2.py", contains the following Python code:

```
1 for i in range(5) :  
2     print('le carré de', i, 'est', i * i)  
3
```

The console window, titled "Console", shows the output of running the script:

```
>>> %Run iterable_range_2.py  
le carré de 0 est 0  
le carré de 1 est 1  
le carré de 2 est 4  
le carré de 3 est 9  
le carré de 4 est 16
```

The status bar at the bottom indicates "Python 3 local • Python de Thonny".

Types itérables en python

Les variables de type *str*, *tuple*, *list*, *range* peuvent être parcourues grâce à l'opérateur **[]** (opérateur d'extraction) :

- en donnant un indice compris entre 0 et *taille* - 1,
- en donnant un indice compris entre -1 et *-taille*.

indices positifs	0	1	2	3	4	5	6
chaîne = B o n j o u r							
indices négatifs	-7	-6	-5	-4	-3	-2	-1

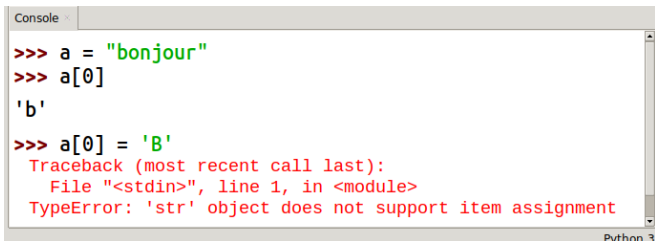
- `chaîne[0]` et `chaîne[-7]` donnent 'B',
- `chaîne[2]` donne 'n', `chaîne[-2]` donne 'u',
- `chaîne[7]` ou `chaîne[-8]` retournent une erreur.

Types immuables en python

Les types *int*, *float*, *bool*, *str* sont **immuables**. On ne peut pas modifier pas la valeur d'une variable de l'un de ces types.

Au mieux, on recrée une nouvelle variable, avec la nouvelle valeur. Tenter de modifier le contenu provoque une erreur.

Exemple avec une variable de type *str* :



```
Console <
>>> a = "bonjour"
>>> a[0]
'b'
>>> a[0] = 'B'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
Python 3
```

Types immuables en python

Le type *tuple* est également immuable.

```
>>> tuple1 = (1, 3, 7)
```

```
>>> type(tuple1)
```

```
<class 'tuple'>
```

```
>>> tuple1[0]
```

```
1
```

```
>>> tuple1[0] = 4
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

Python 3 loca

Types muables en python

Comme le type **tuple**, les types **list**, **dict**, **set** permettent de structurer les données.

MAIS, contrairement au type **tuple**, ces types sont *muables*.

Les variables de types muables peuvent être modifiées par des *méthodes* qu'on leur applique.

Ces méthodes sont plus efficaces qu'une affectation car elles **ne changent pas l'emplacement mémoire**. On ne perd donc pas le temps nécessaire à la création de la variable.

Les méthodes sont appelées par la *notation pointée* :

- `l1.append(17)` pour ajouter l'élément 17 à la liste `l1`
- `l1.sort()` pour trier la liste `l1`

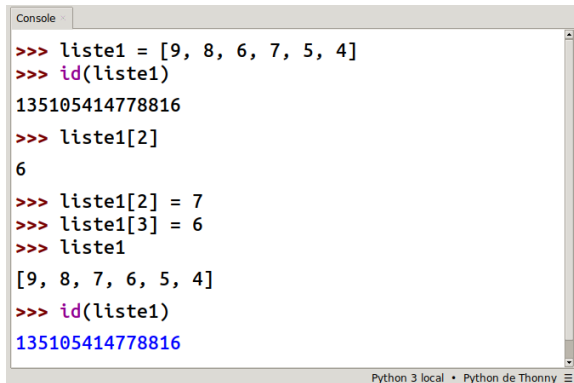
Types muables en python

```
>>> liste1 = [24, 35, 27]
>>> print(liste1, "...", id(liste1))
[24, 35, 27] ... 128537939707456
>>> liste1.append(17)
>>> print(liste1, "...", id(liste1))
[24, 35, 27, 17] ... 128537939707456
>>> liste1.sort()
>>> print(liste1, "...", id(liste1))
[17, 24, 27, 35] ... 128537939707456
```

Types muables en python

Les variables de types muables peuvent aussi être modifiées en utilisant l'opérateur [].

Par exemple, on peut modifier des éléments d'une liste :

A screenshot of a Python console window titled 'Console'. It displays a series of commands and their outputs. The commands are: creating a list 'liste1' with values [9, 8, 6, 7, 5, 4], getting its ID, accessing the element at index 2, and then modifying the elements at indices 2 and 3. The final output shows the modified list and its ID, which remains the same as before the modifications.

```
>>> liste1 = [9, 8, 6, 7, 5, 4]
>>> id(liste1)
135105414778816
>>> liste1[2]
6
>>> liste1[2] = 7
>>> liste1[3] = 6
>>> liste1
[9, 8, 7, 6, 5, 4]
>>> id(liste1)
135105414778816
```

Python 3 local • Python de Thonny

Un exemple comparatif

On cherche à construire une liste des n premiers entiers positifs, de 0 à $n - 1$.

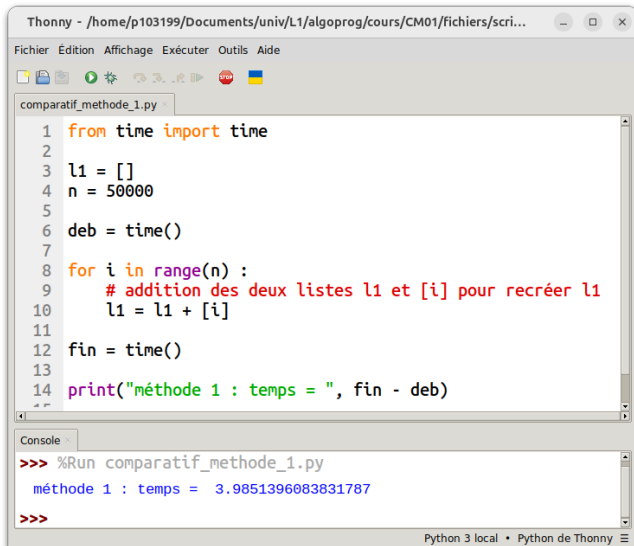
Pour cela, on propose deux manières de procéder :

- Partir de la liste vide `[]`, et lui ajouter, par l'opérateur `+` et une affectation, des listes de 1 élément, ceci n fois.
- Partir de la liste `[]`, et lui ajouter, par la méthode *append* spécifique aux listes, 1 élément à la fois, ceci n fois.

On mesure le temps d'exécution de chaque méthode pour comparer leur efficacité.

Un exemple comparatif

Résultat de la méthode 1



The screenshot shows the Thonny Python IDE interface. The main window displays a Python script named `comparatif_methode_1.py` with the following code:

```
1 from time import time
2
3 l1 = []
4 n = 50000
5
6 deb = time()
7
8 for i in range(n):
9     # addition des deux listes l1 et [i] pour recréer l1
10    l1 = l1 + [i]
11
12 fin = time()
13
14 print("méthode 1 : temps = ", fin - deb)
```

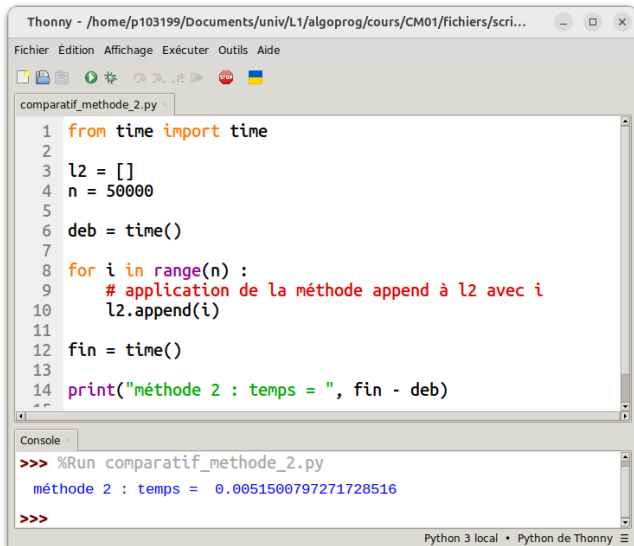
Below the script editor is a console window showing the execution of the script:

```
>>> %Run comparatif_methode_1.py
méthode 1 : temps = 3.9851396083831787
>>>
```

The status bar at the bottom indicates "Python 3 local" and "Python de Thonny".

Un exemple comparatif

Résultat de la méthode 2



```
Thonny - /home/p103199/Documents/univ/L1/algoprogram/cours/CM01/fichiers/scri...
Fichier Édition Affichage Exécuter Outils Aide

comparatif_methode_2.py
1 from time import time
2
3 l2 = []
4 n = 50000
5
6 deb = time()
7
8 for i in range(n):
9     # application de la méthode append à l2 avec i
10    l2.append(i)
11
12 fin = time()
13
14 print("méthode 2 : temps = ", fin - deb)

Console
>>> %Run comparatif_methode_2.py
méthode 2 : temps = 0.0051500797271728516
>>>
```

Python 3 local • Python de Thonny

Un exemple comparatif

Explications

- L'instruction `l1 = l1 + [i]` est une affectation qui fabrique une nouvelle variable : gestion de l'espace mémoire, temps consommé ;
- L'instruction `l2.append(i)` agit toujours sur le même emplacement mémoire, temps gagné.
- Ce gain de temps a été possible car le type *list* est un type dont les valeurs sont *modifiables* ou *muables*.
- Certains types sont *immuables*. Ces modifications au même emplacement mémoire sont alors impossibles.

Types muables en python

On peut aussi modifier une variable de type `set` par des méthodes spécifiques (`add`, `discard` par exemple)

```
>>> fruits = {"banane", "ananas", "citron", "kiwi", "abricot"}
>>> fruits
{'ananas', 'kiwi', 'abricot', 'banane', 'citron'}
>>> id(fruits)
125793888056960
>>> fruits.add("fraise")
>>> fruits
{'ananas', 'kiwi', 'abricot', 'banane', 'fraise', 'citron'}
>>> id(fruits)
125793888056960
>>> fruits.discard("abricot")
>>> fruits
{'ananas', 'kiwi', 'banane', 'fraise', 'citron'}
>>> id(fruits)
125793888056960
```

Types muables en python

Une variable de type **dict** est un tableau de paires clé/valeur.
On peut définir de nouvelles associations ou en supprimer.

```
>>> d1 = {'x' : 5, 'y' : 2}
>>> type(d1)
<class 'dict'>
>>> id(d1)
125793883122368
>>> d1['x']
5
>>> d1['z'] = 3.4
>>> id(d1)
125793883122368
>>> d1
{'x': 5, 'y': 2, 'z': 3.4}
```