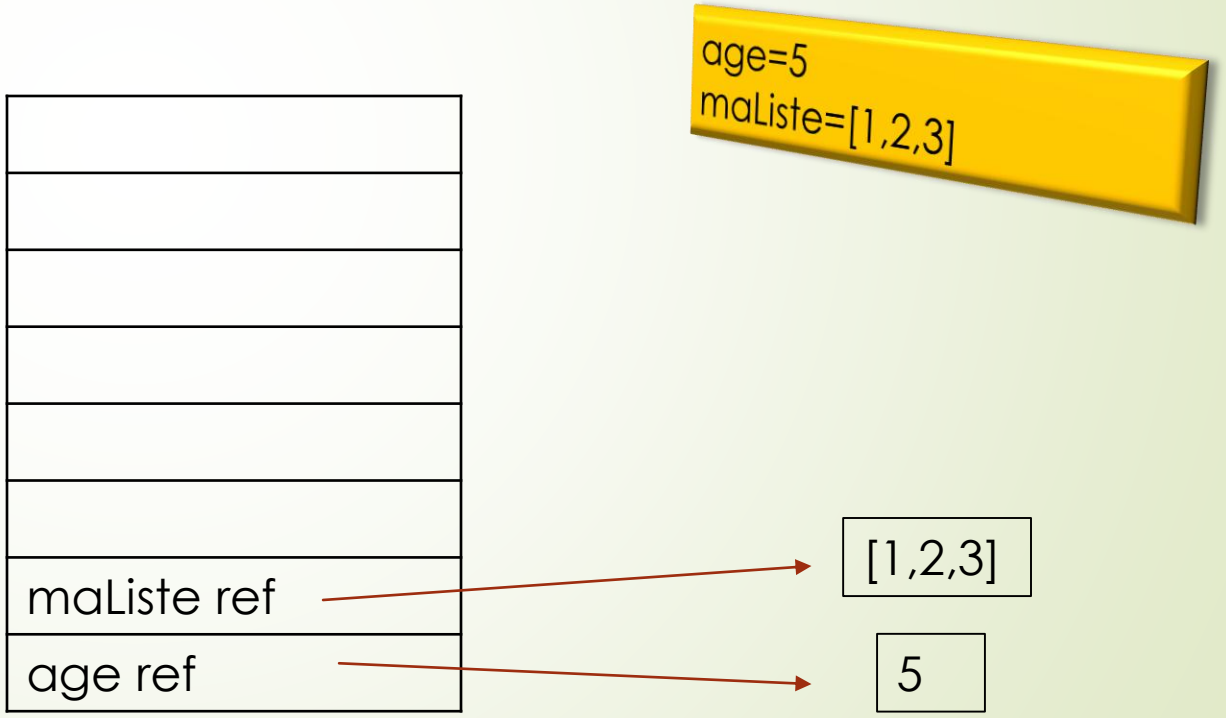
Quelques rappels

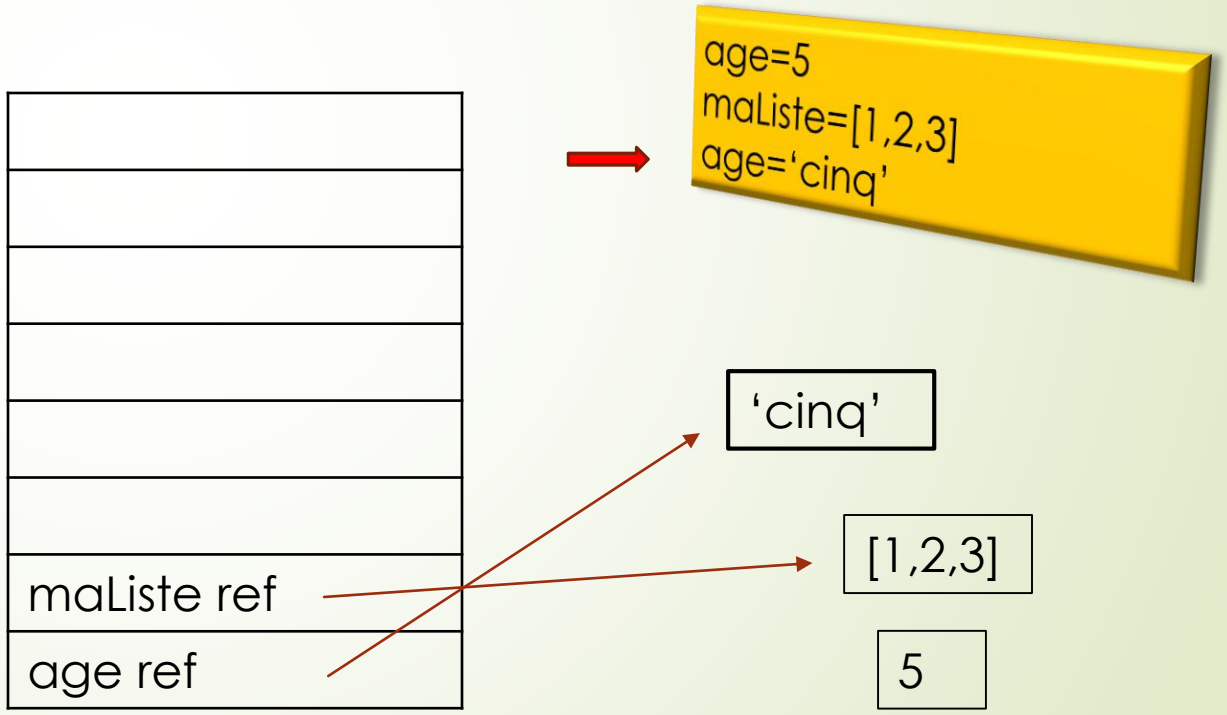
Rappels

En Python les variables sont des références à des objets



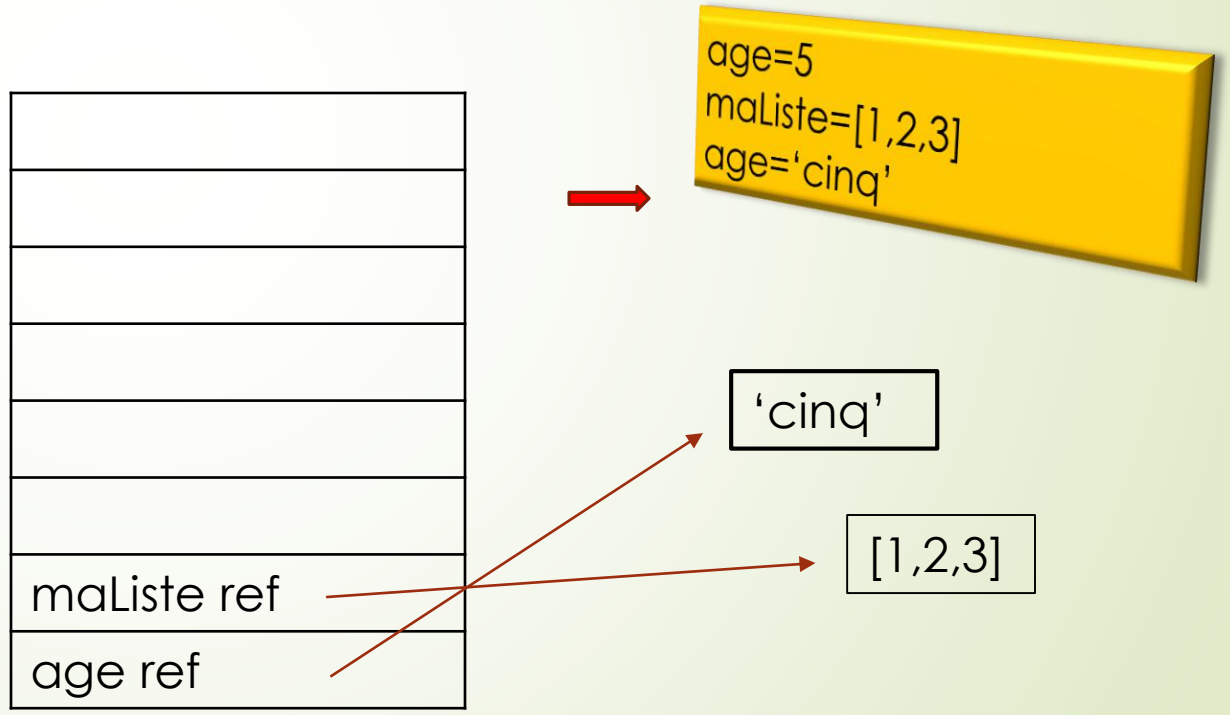
Rappels

Création d'un nouvel objet, l'ancien n'est plus référencé



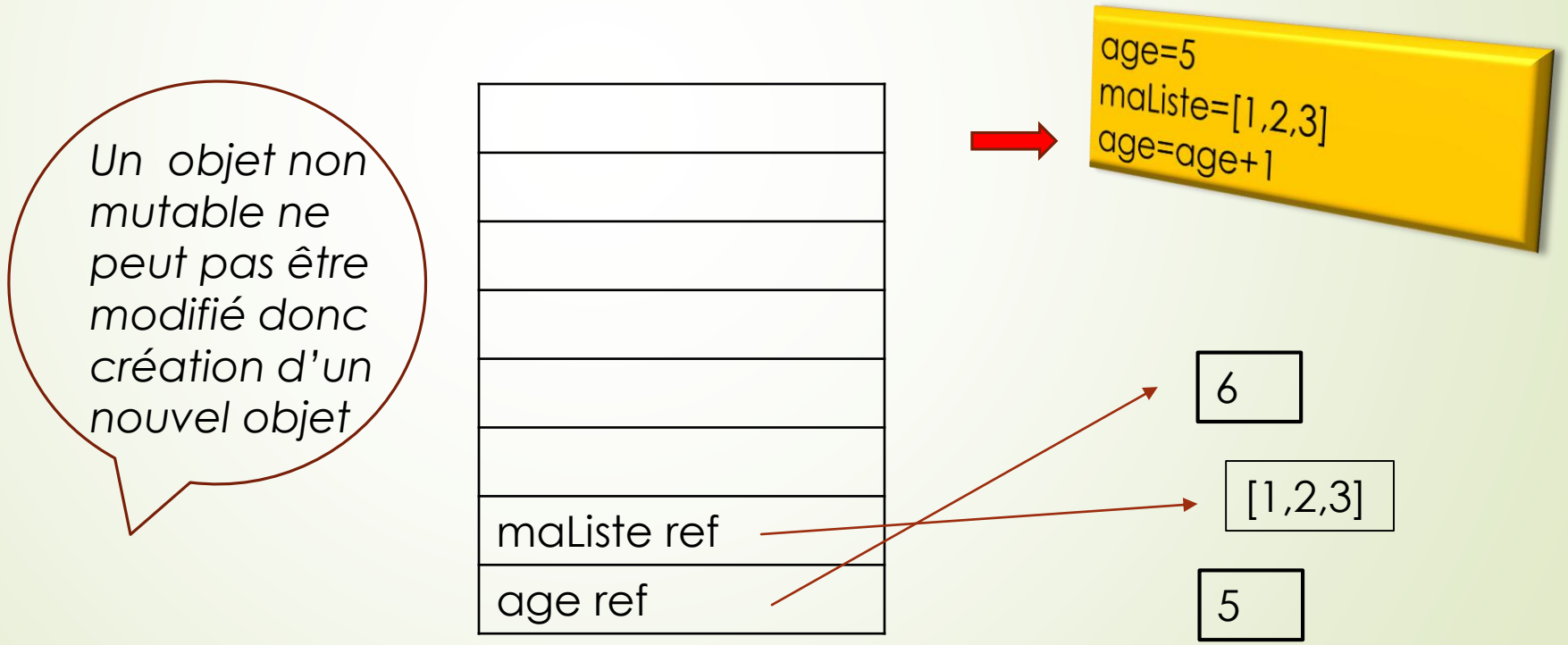
Rappels

Un objet qui n'est plus référencé est retiré de la mémoire (garbage collector)



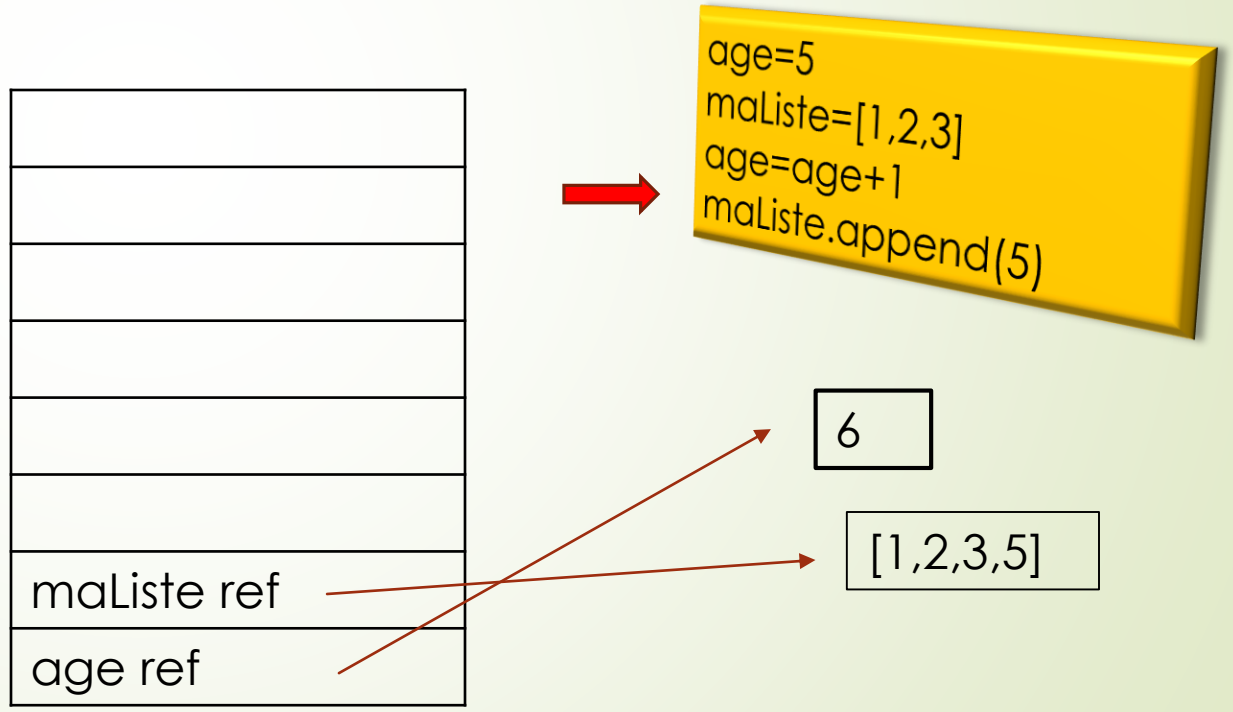
Rappels

Un objet non mutable ne peut pas être modifié donc création d'un nouvel objet



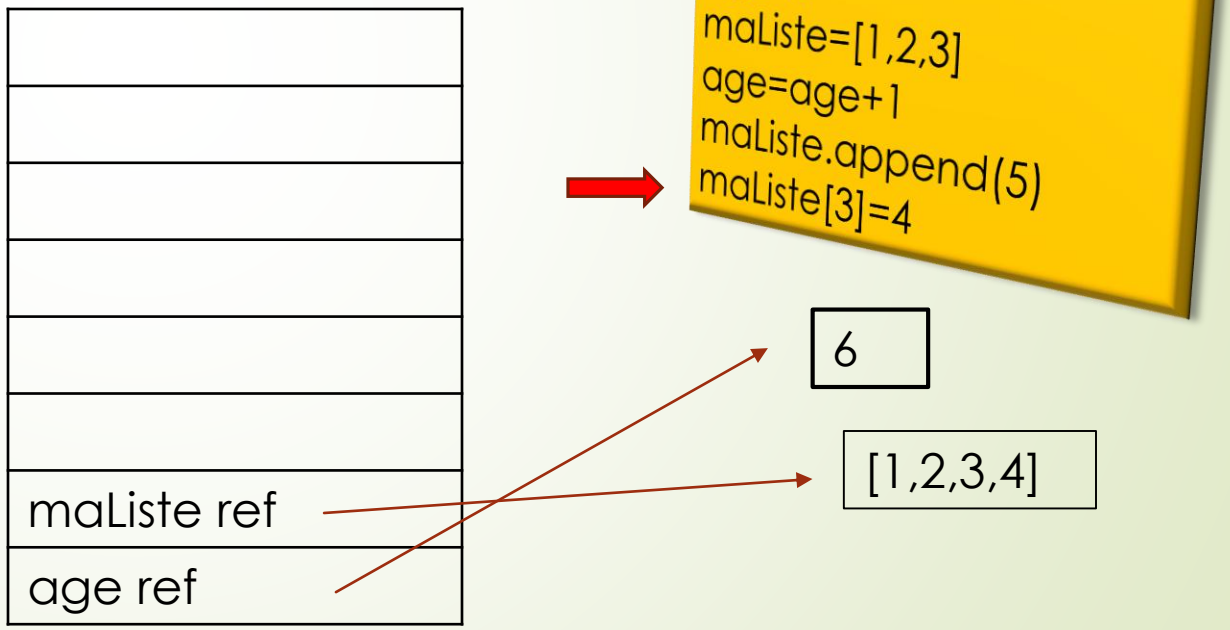
Rappels

Un objet mutable peut être modifié

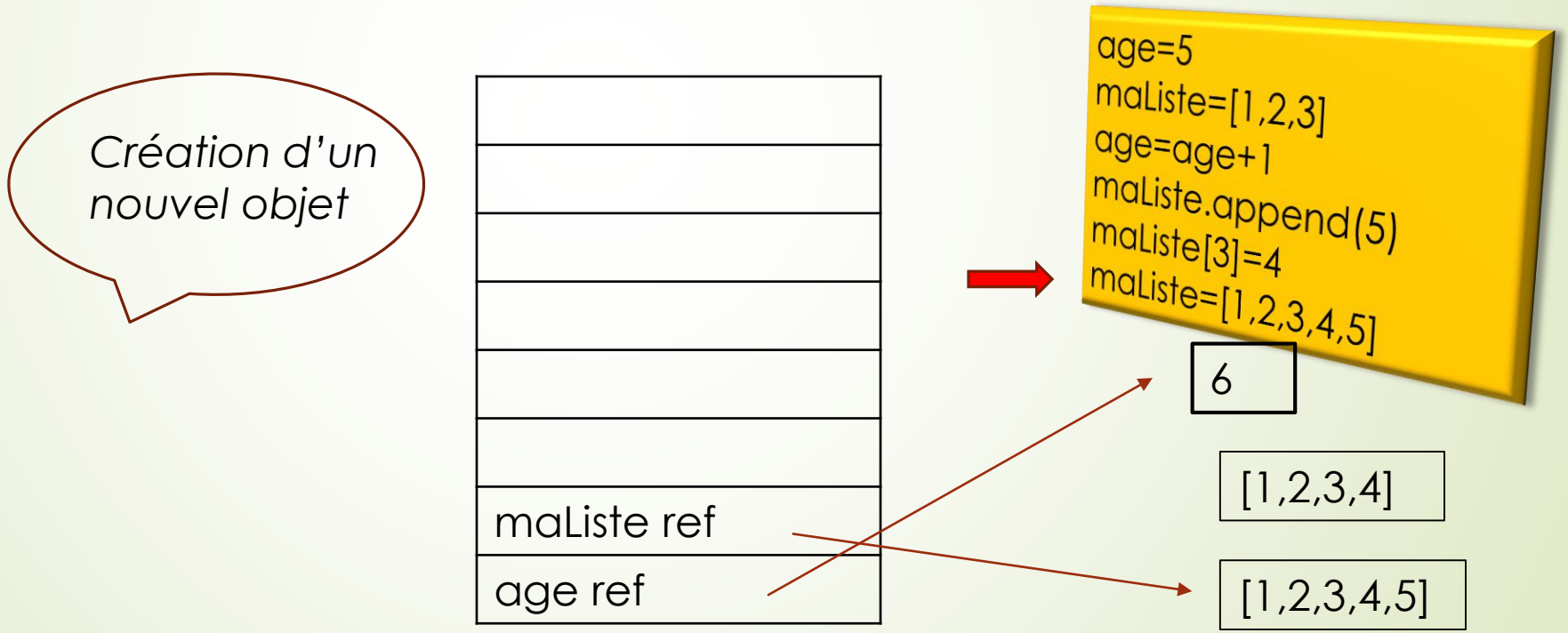


Rappels

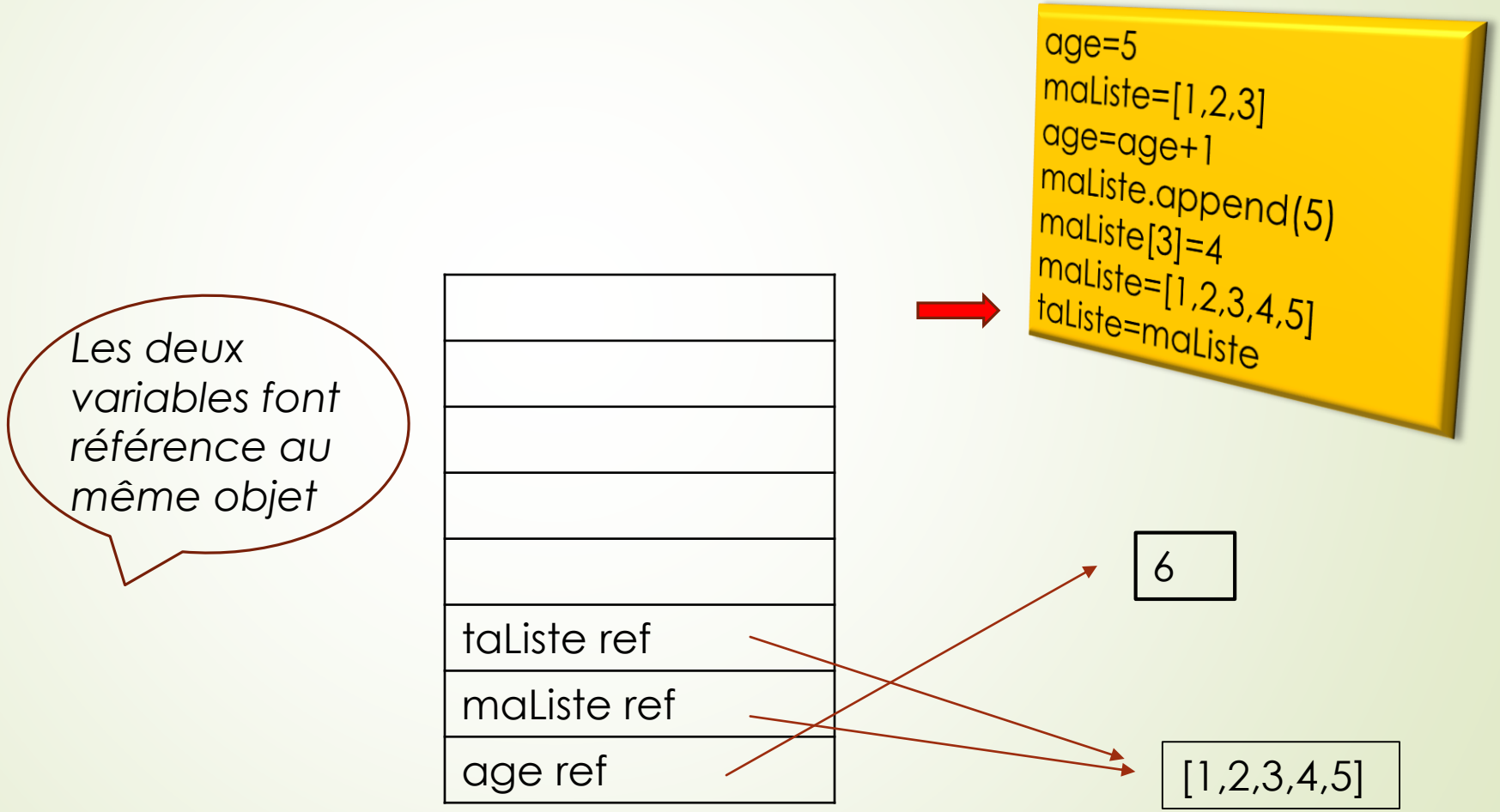
Un objet mutable peut être modifié



Rappels

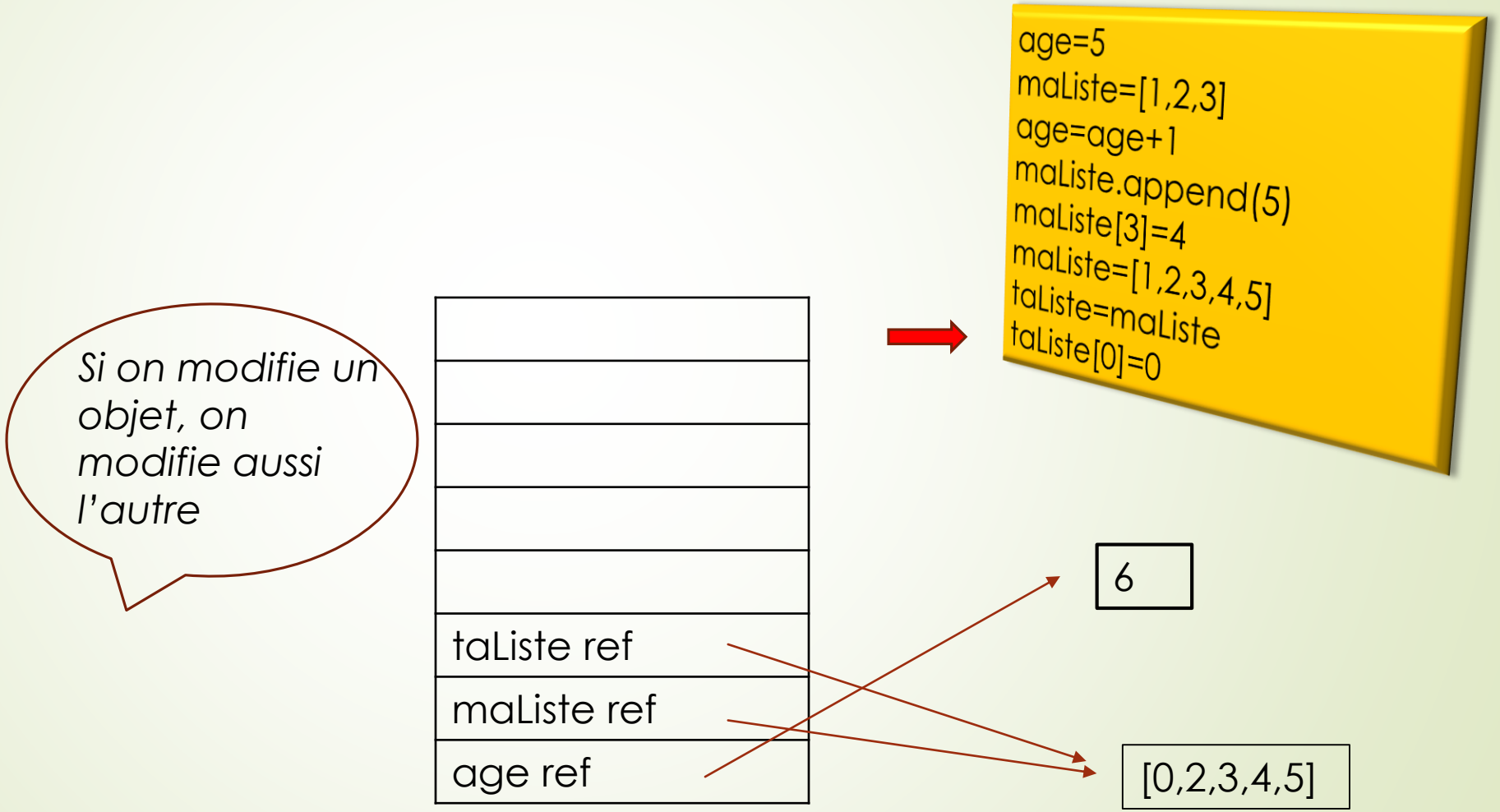

Création d'un nouvel objet
maListe ref
age ref
age=5
maListe=[1,2,3]
age=age+1
maListe.append(5)
maListe[3]=4
maListe=[1,2,3,4,5]
6
[1,2,3,4]
[1,2,3,4,5]

Rappels


age=5
maListe=[1,2,3]
age=age+1
maListe.append(5)
maListe[3]=4
maListe=[1,2,3,4,5]
taListe=maListe
Les deux variables font référence au même objet
taListe ref
maListe ref
age ref
6
[1,2,3,4,5]

Rappels

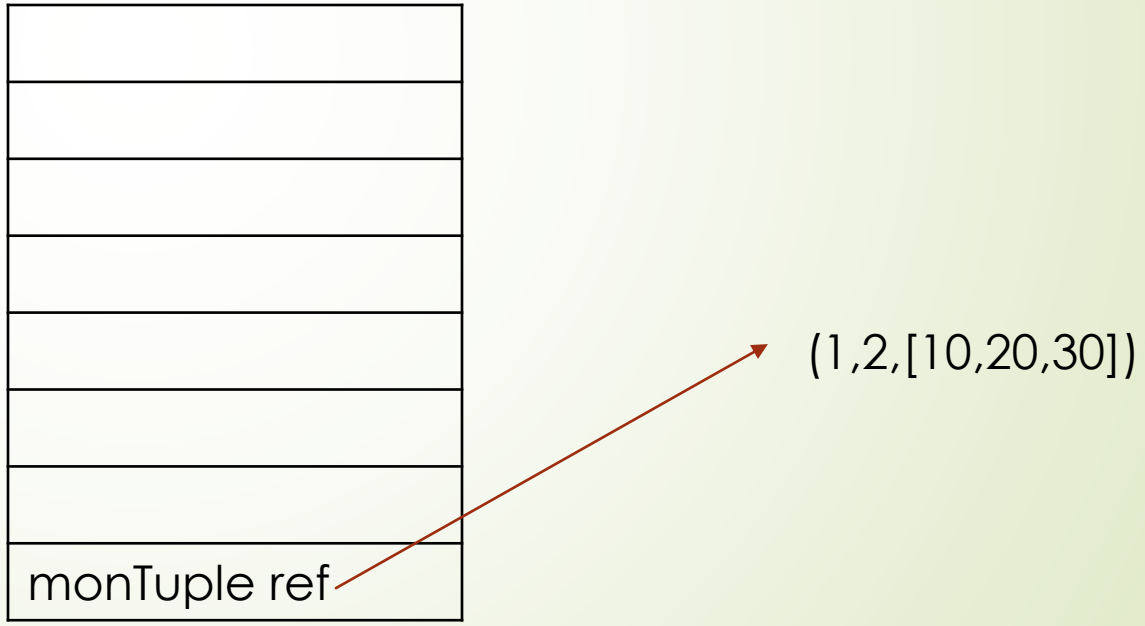
Si on modifie un objet, on modifie aussi l'autre



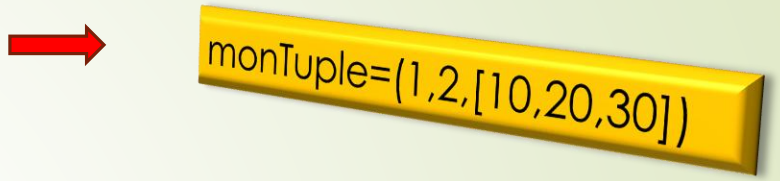
Rappels

monTuple=(1,2,[10,20,30])

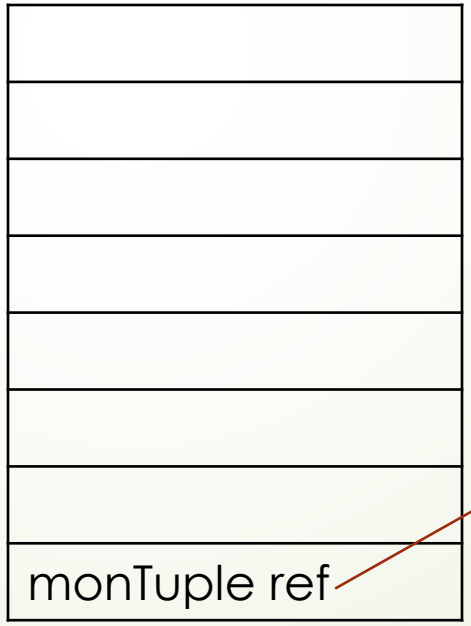
Un tuple est non mutable On n'a pas le droit de modifier son contenu

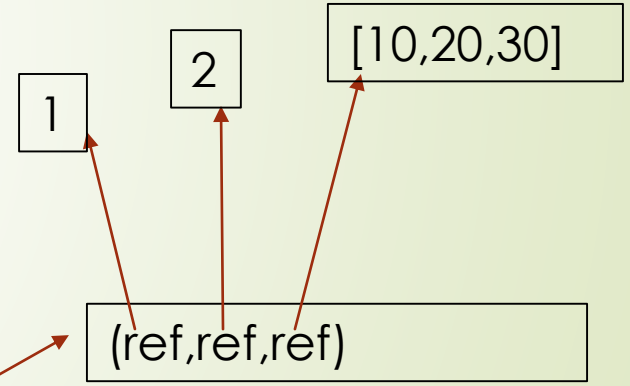


Rappels



En fait, on n'a pas le droit de modifier les références qu'il contient

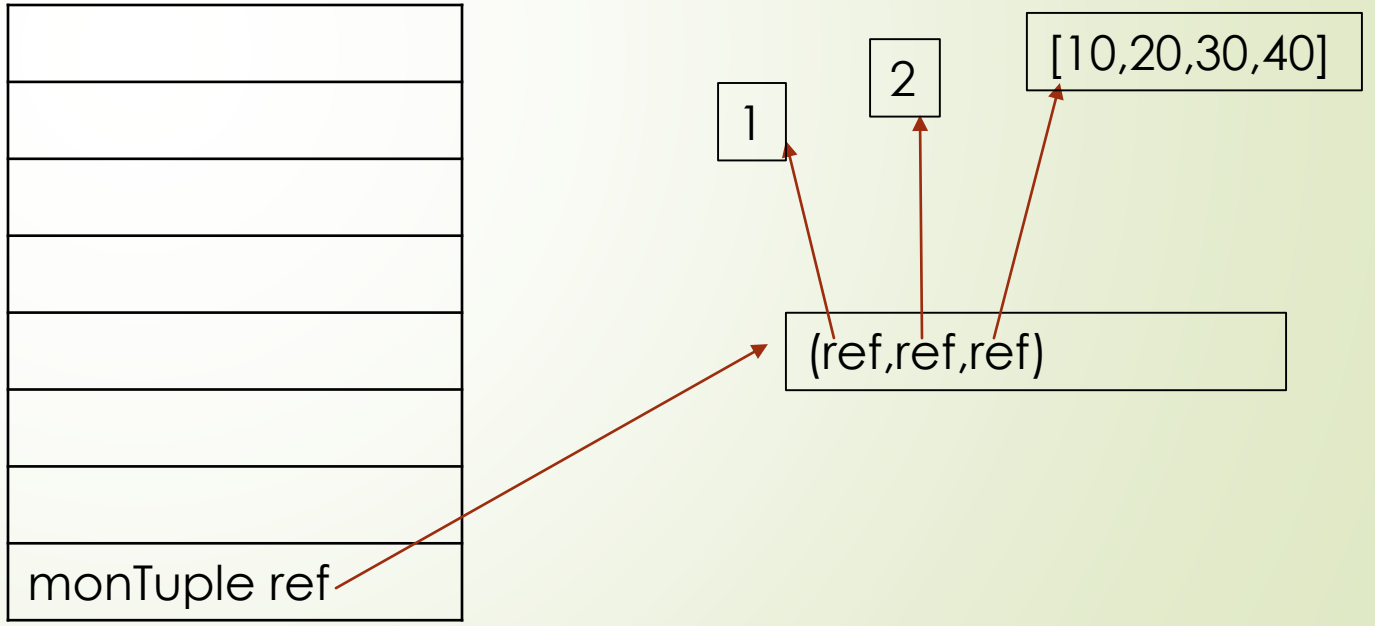




Rappels

monTuple=(1,2,[10,20,30])
monTuple[2].append(40)

Mais on peut modifier l'objet à cette référence s'il est mutable



Mutable/ non mutable

- Types mutables
  - List
  - Dict
  - set
- Types non mutables
  - int
  - Float
  - Complex
  - String
  - Tuple
  - Range
  - frozenset

fonctions

Programmation structurée:Passage de paramètres

- Lors de la définition d'une fonction les paramètres sont les paramètres formels: leur nom importe peu ce n'est qu'un formalisme qui permet de décrire le comportement de la fonction

- Lors de l'appel de la fonction, les paramètres sont dits effectifs car c'est effectivement sur eux que la fonction va s'exécuter.

Par adresse et par valeur

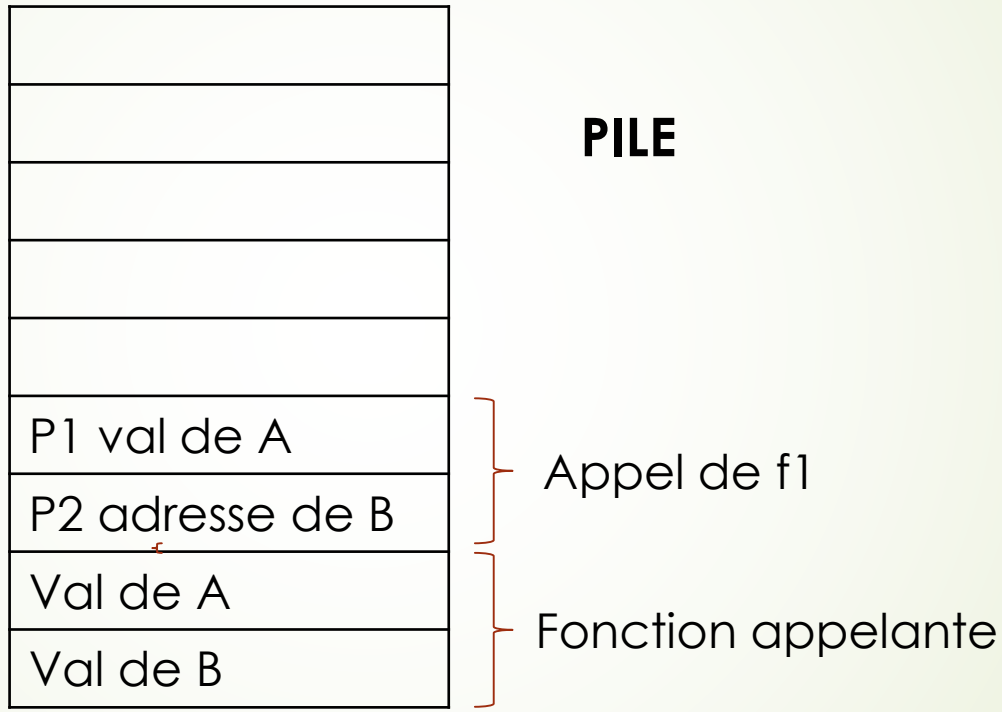
- En algorithmique il existe deux passages de paramètres: par valeur et par adresse (ou référence)
- Soit une fonction f1 avec deux paramètres: p1 passé par valeur et p2 passé par adresse
  - F1(param p1, param p2)
  - Début

    p1=T1(p1)

    P2=T2(p2)
  - Fin

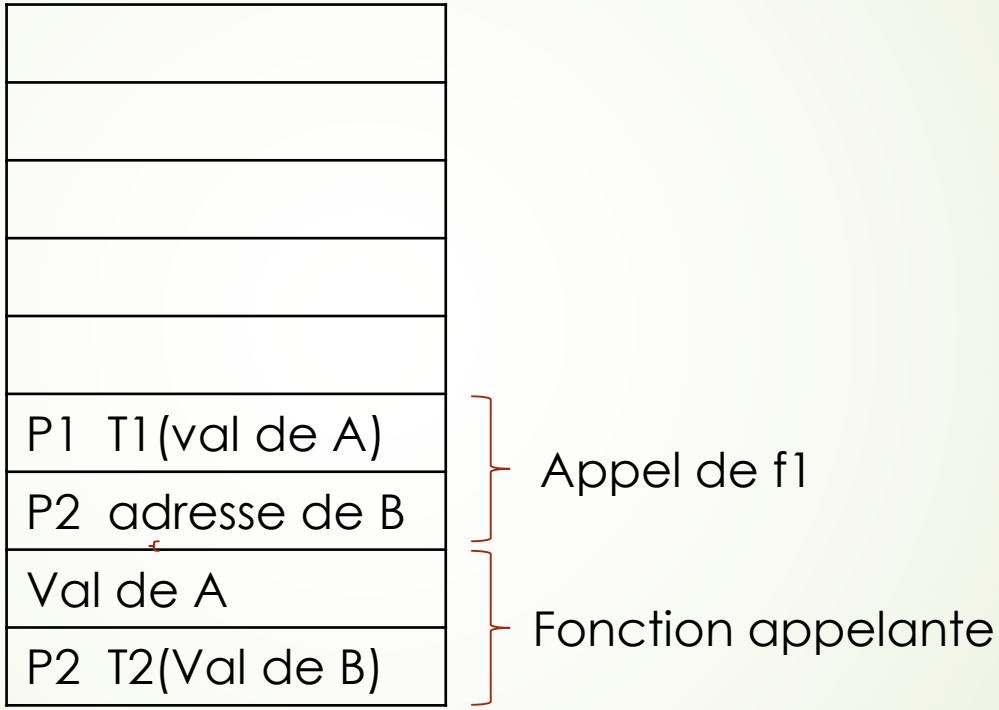
Appel de la fonction

- ...
- F1(A,B)



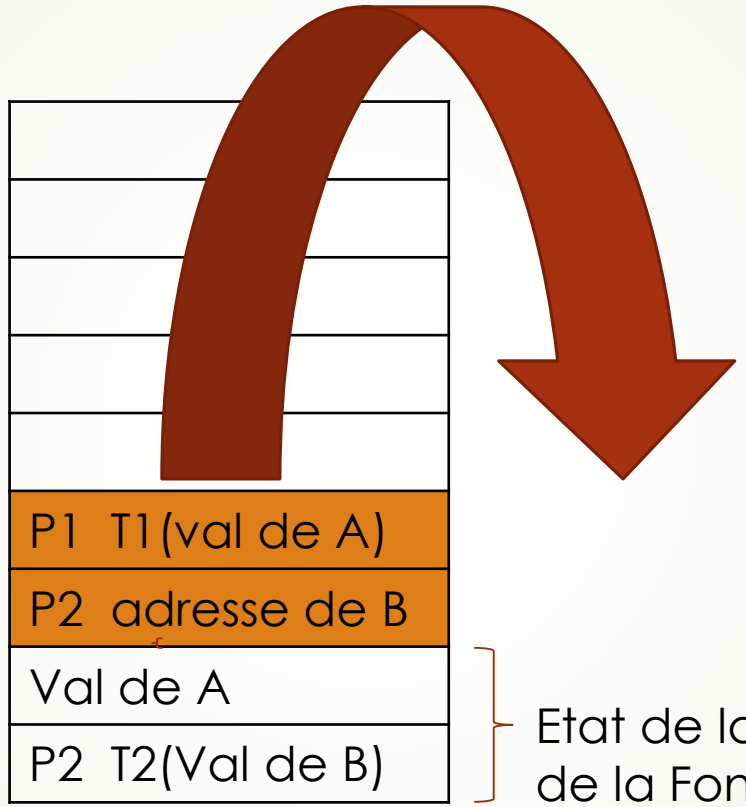
exécution de la fonction

- …
- F1(A,B)



Après exécution de la fonction

- …
- F1(A,B)

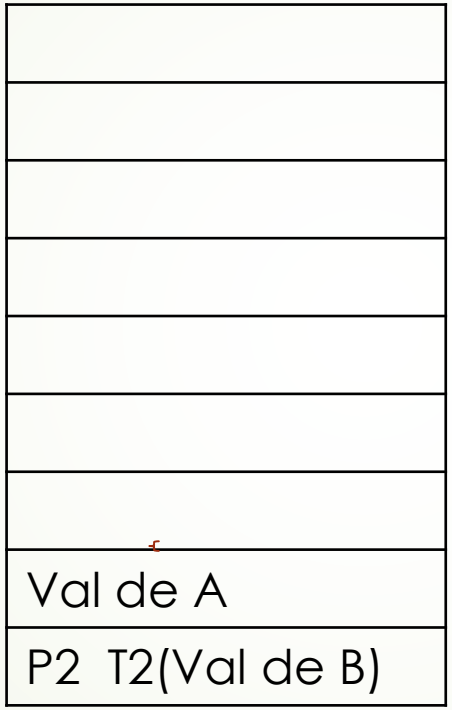


On dépile la zone de l'exécution de la fonction

Etat de la mémoire de la Fonction appelante

Conclusion

- …
- F1(A,B)



Le paramètre passé par valeur n'a pas changé

Pour le paramètre passé par adresse, les changements ayant été faits à l'adresse indiquée ils sont sauvegardés

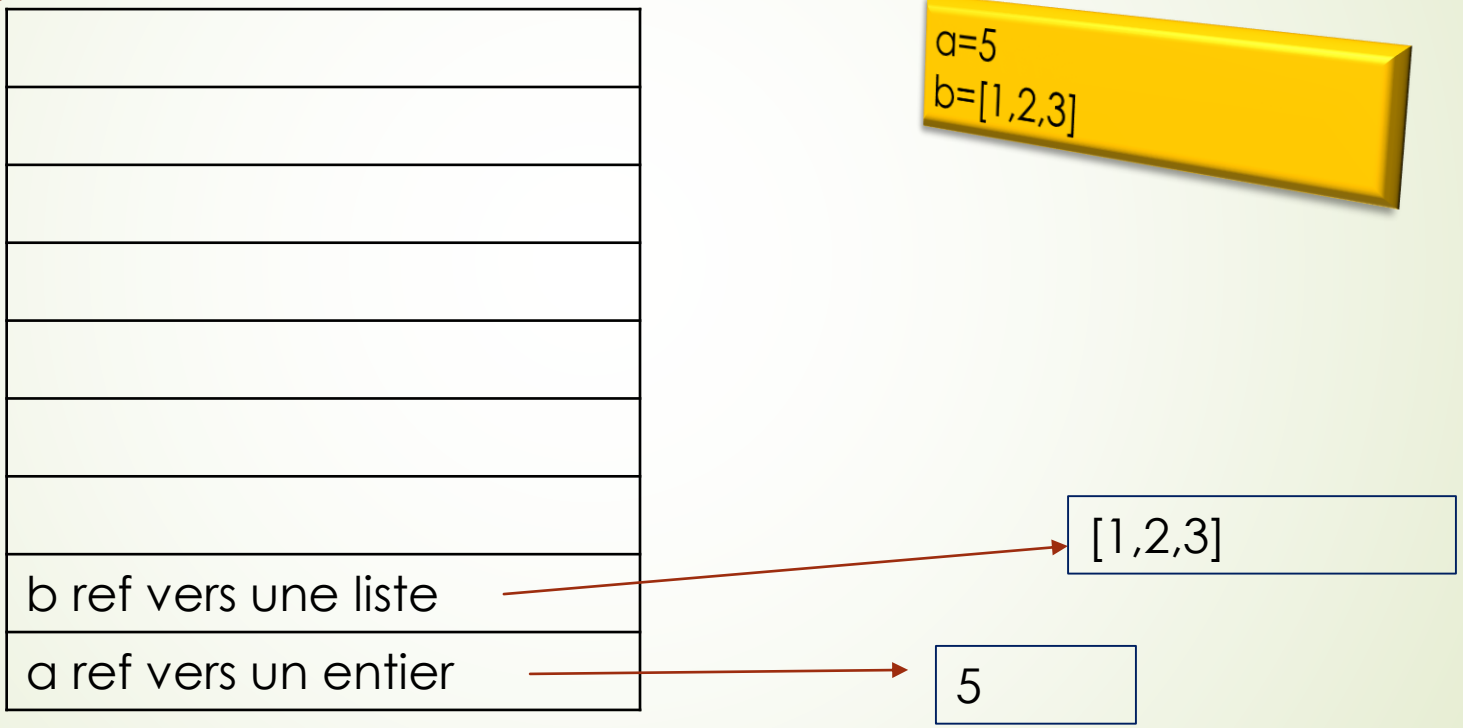
Passage de paramètres en Python

- Quand on passe un paramètre en Python on passe une référence (passage par adresse)
- Si on passe un entier étant donné que c'est un type non mutable: si on ne change pas la référence on ne peut pas modifier l'entier.
- Si on passe une liste , on récupérera la même référence à la fin de la fonction mais étant donné que c'est un type mutable des modifications peuvent avoir été faites à cette référence et donc elles sont récupérées.
- Quand on fait un return on récupère une nouvelle référence

- Quand on utilise une valeur pas défaut on utilise une référence qui reste la même

Exemple (1)

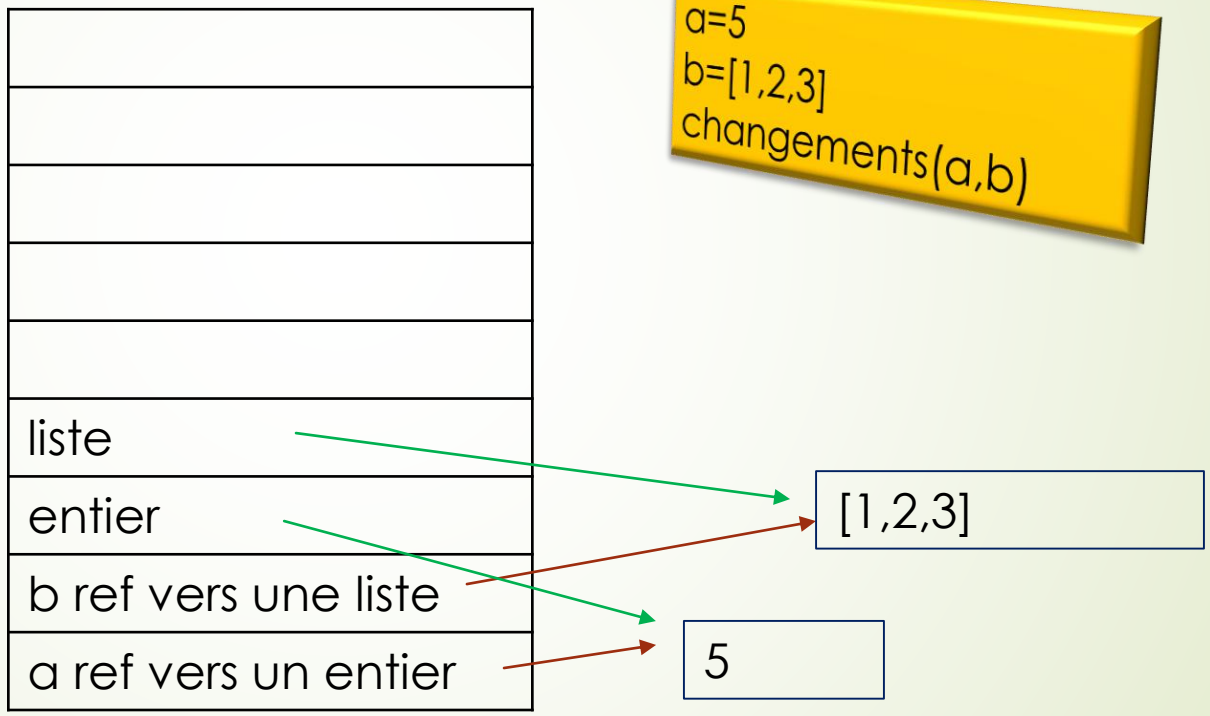
Stockage des variables dans la pile



Exemple(2)

```
def changements(entier, liste):
    entier=entier+1
    liste.append(4)
```

On empile l'espace nécessaire à la fonction: on donne à chaque paramètre la référence contenue dans le paramètre effectif correspondant

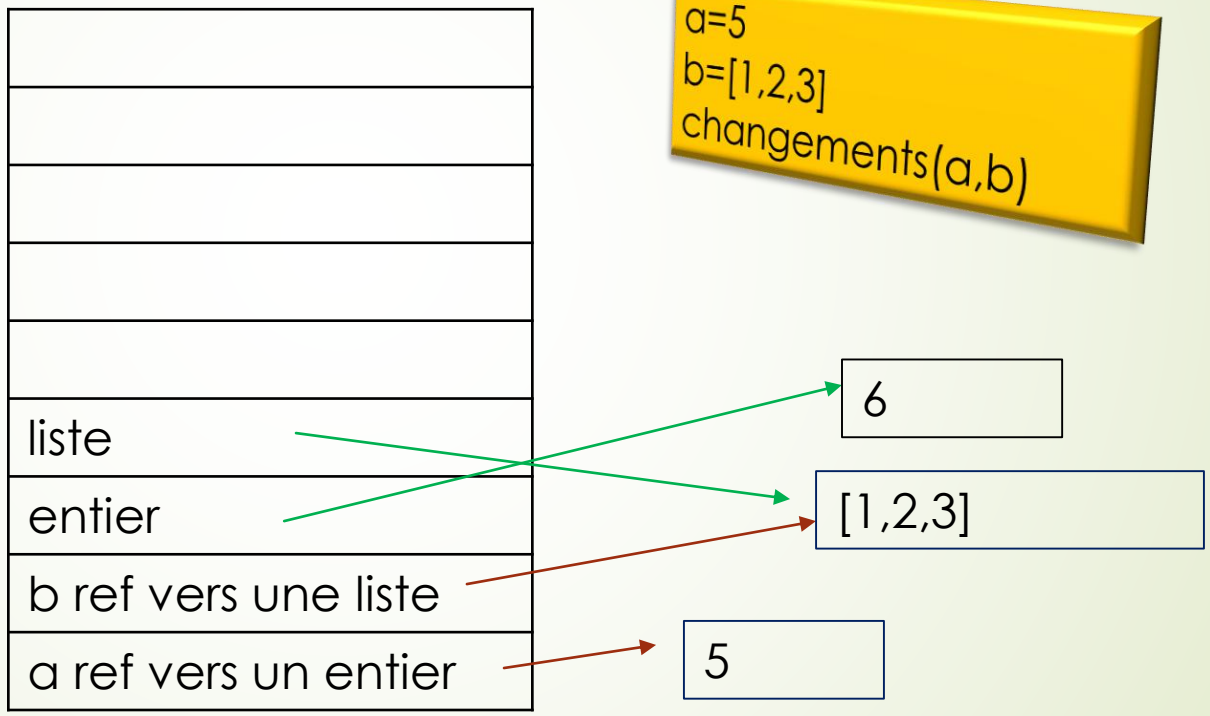



```
a=5
b=[1,2,3]
changements(a,b)
```

Exemple(3)

```
def changements(entier, liste):
    entier=entier+1
    liste.append(4)
```

Type entier non mutable: création d'un nouvel entier

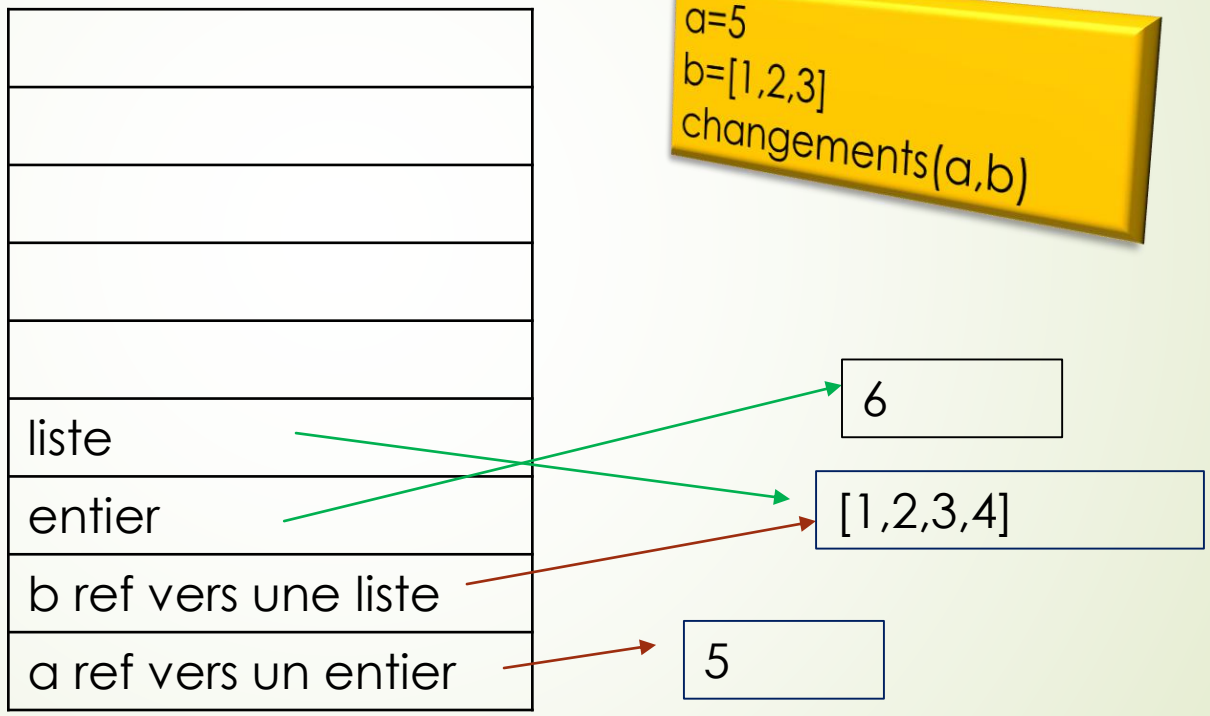



```
a=5
b=[1,2,3]
changements(a,b)
```

Exemple(4)

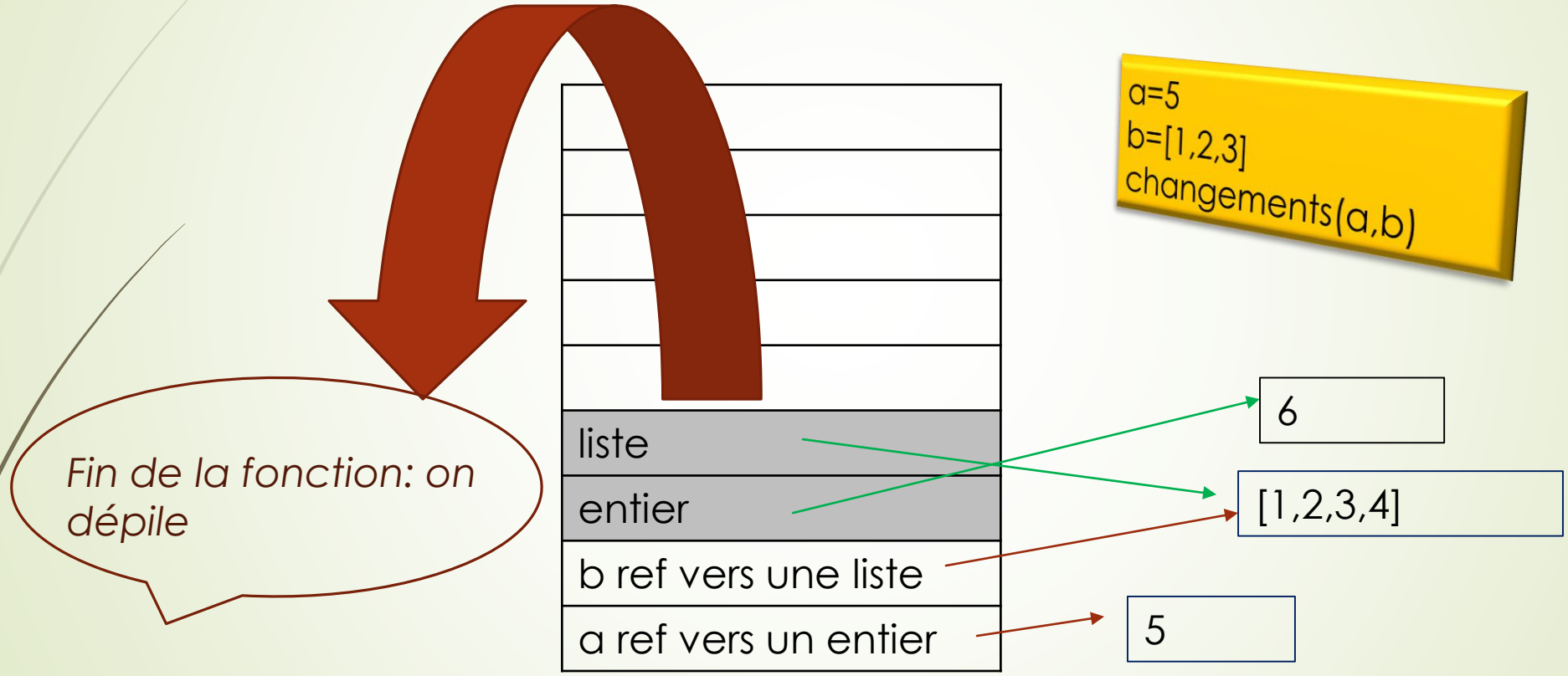
```
def changements(entier, liste):
    entier=entier+1
    liste.append(4)
```

Type liste mutable: modification de la liste existante

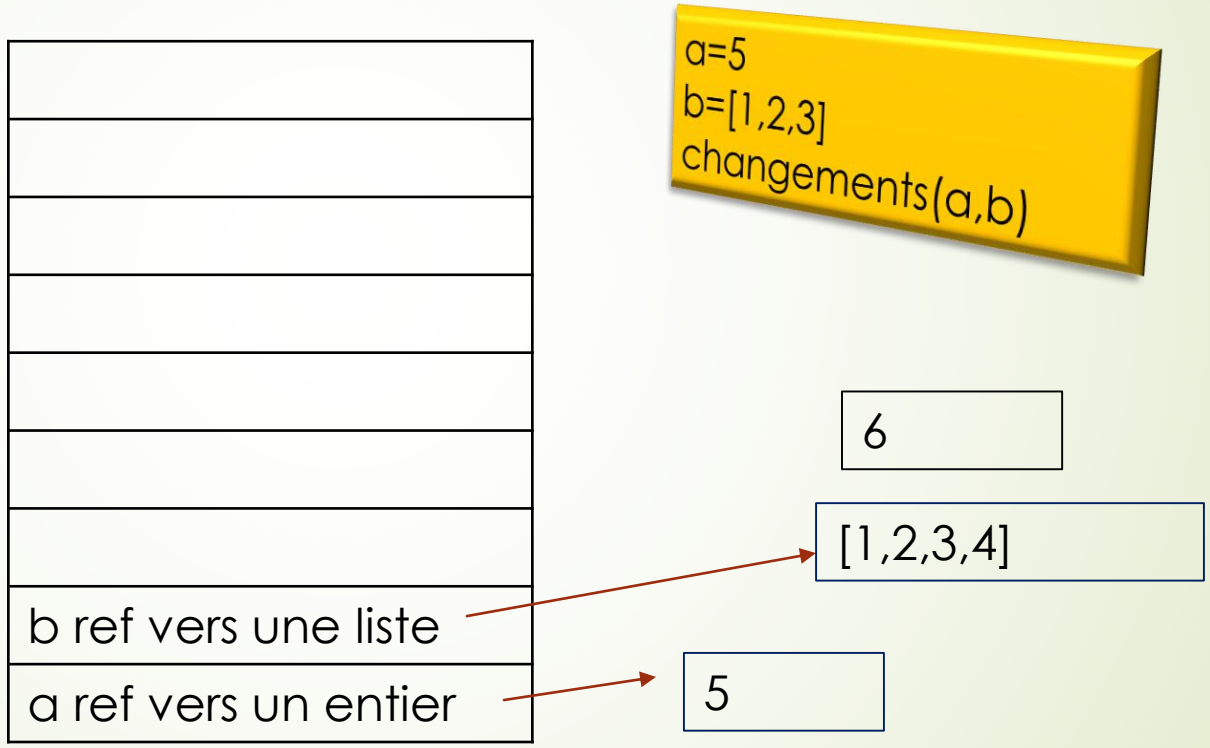



```
a=5
b=[1,2,3]
changements(a,b)
```

Exemple(5)

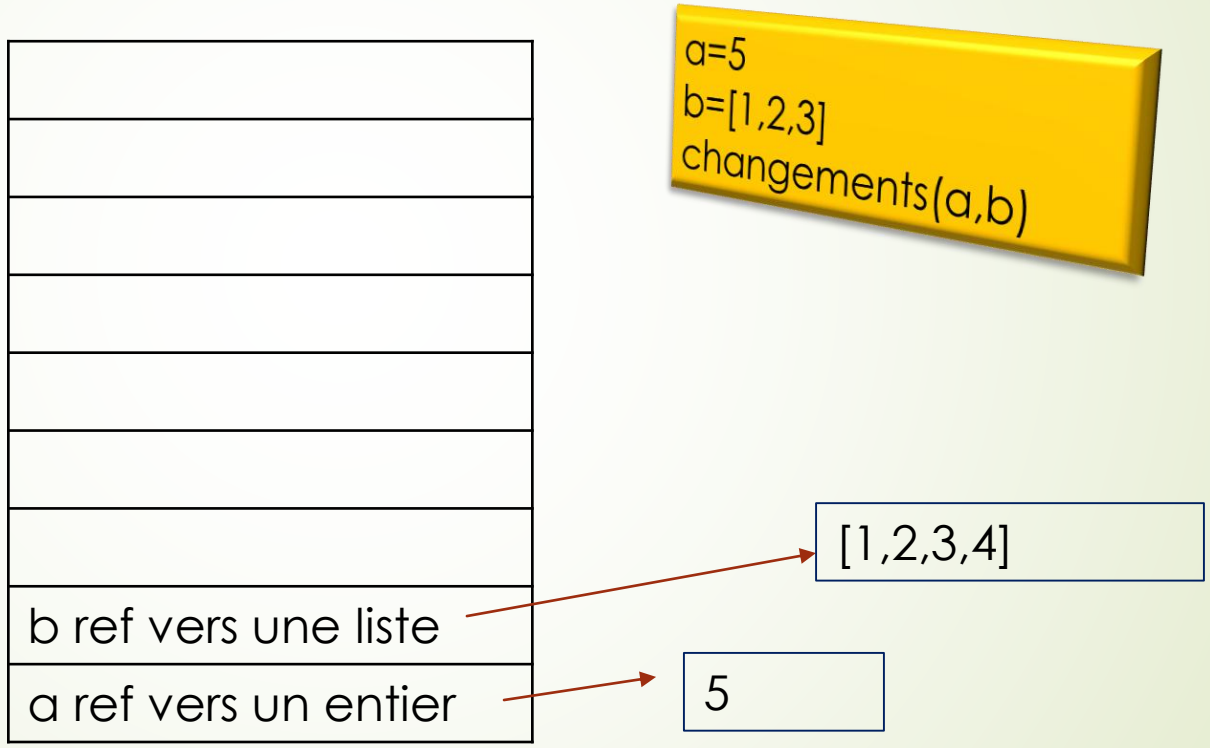

a=5
b=[1,2,3]
changements(a,b)
liste
entier
b ref vers une liste
a ref vers un entier
6
[1,2,3,4]
5
Fin de la fonction: on dépile

Exemple(6)



L'entier 6 n'est plus adressé le garbage collector va le supprimer

Exemple(7)



Après la fin de l'exécution de la fonction: changements gardés pour le paramètre mutable et pas pour le paramètre non mutable

Module/ package

- Un module est un fichier .py contenant ayant un contenu importable ou utilisable
- Importation du module: import
  - Import math
- Ce qui est défini (fonction_1)dans le module mon_module est accessible par :

  Mon_module.fonction_1

  math.pi
- Alias: import math as m: m.pi
- Savoir ce qu'il y a dans un module ; help() et dir()

Module : importation ciblée

- importation ciblée
  - From math import pi

- Importation de tout ce qui est défini dans le module
  - From math import *

  AVEC FROM ON NE SPECIFIE PAS LE NOM DU MODULE A L'UTILISATION

  Importation de plusieurs modules
  - Import math
  - Import turtle
  - Import math, turtle

  SANS FROM ON SPECIFIE LE NOM DU MODULE

Avec import *module*
, le module lui-meme est importé mais il garde son propre espace de noms,
c'est pourquoi vous devez utiliser le nom du module pour accéder à ses fonctions ou attributs :
module.fonction.
Mais avec from *module* import,
vous importez des fonctions et des attributs spécifiques d'un autre module
dans votre propre espace de noms,
c'est pourquoi vous y accédez directement sans référence au module dont ils viennent

Importation?

- Importation des fonctions et aussi exécution du code: peut être gênant
- Si on veut un bloc d'instructions qui ne soient pas executée lors de l'importation:
  - If __name__=='__main':

    Bloc d'instructions

Package

- Un Package est un
- Contient le fichier __ini__.py
- From package.module import function

Espaces de noms

- Un espace de noms est comme un dictionnaire :
  - clés = noms des variables
  - valeurs =valeurs des variables

- **La portée globale** : celle du module ou du fichier script en cours. Un dictionnaire gère les objets globaux :
  - l'instruction globals() fournit un dictionnaire contenant les couples nom:valeur ;
- **La portée locale** : les objets internes aux fonctions sont locaux. Les objets globaux ne sont *pas modifiables*
  - dans les portées locales. L'instruction locals() fournit un dictionnaire contenant les couples nom:valeur.

- La recherche des noms est d'abord locale (L), puis globale (G), enfin interne (I) :

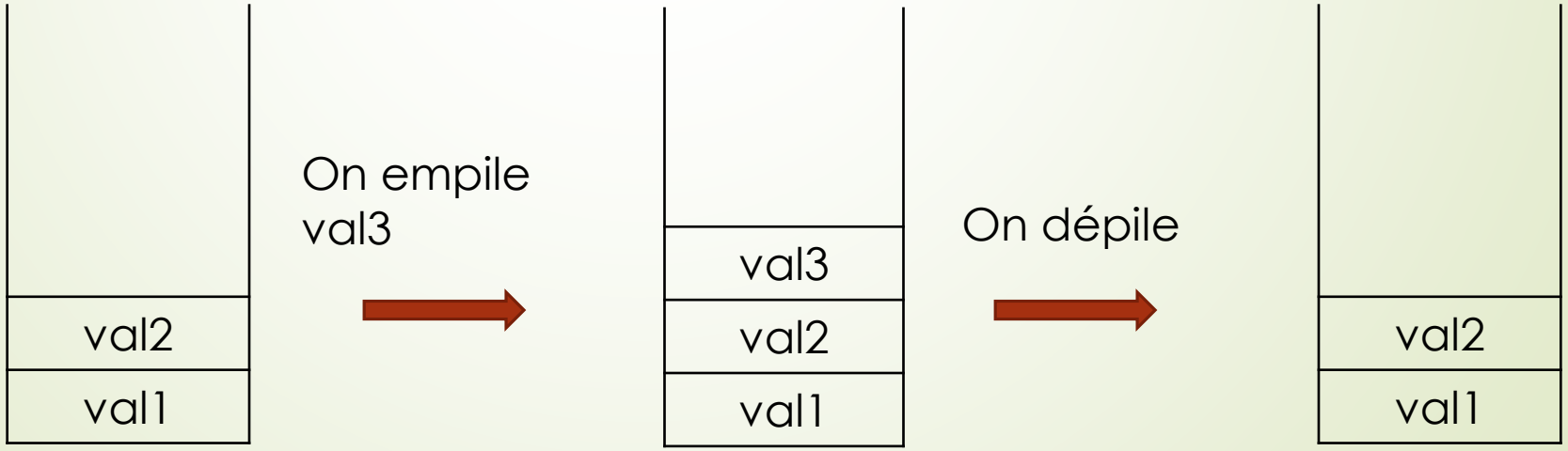
- **Interne**
  - ***Noms prédéfinis* : open, len,...**
- **Global**
  - ***Noms affectés à la base d'un module***
  - ***Noms déclarés* global dans *une fonction ou un module***
- **Local**
  - ***Noms affectés dans une fonction ou un module:*** variables locales et arguments

Fonctions / méthodes

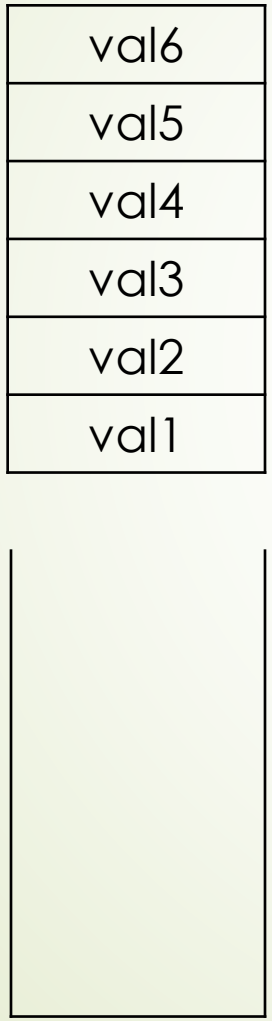
- Une méthode contient des instructions comme une fonction
- Une méthode s'applique à un objet
- Syntaxe de l'appel: objet.nomMéthode(paramètres)
- Différence importante: étant donné que la méthode connait l'objet auquel elle s'applique, il n'est pas nécessaire qu'elle le passe en paramètres Conclusion: une méthode a toujours un paramètre de moins que le fonction qui ferait la même chose

Pile, stack, LIFO (Last In First Out)

Pour ne pas faire de casse:
On ajoute une assiette au-dessus des autres ***on empile(push)***
On enlève l'assiette qui se trouve en haut de la pile ***on dépile (pop)***



Pile, stack, LIFO (Last In First Out)



*On doit savoir quand la **pile est pleine**:*
On ne peut plus empiler

*On doit savoir quand la **pile est vide**:*
On ne peut plus dépiler

Pile, stack, LIFO (Last In First Out)

- Algorithmes correspondants aux 4 opérations

```
Fonction empiler(Pile p, entier cpt, element elt)
Début
        si non pilepleine(P) alors
        début
                cpt<-cpt+1
                p[cpt]<- elt
        finsi
Fin
```

```
Fonction dépiler(Pile p, entier cpt): element
Var locale
Début
        si non pilevide(P) alors
        début
                locale<-p[cpt]
        finsi
        Retourne locale
Fin
```

Pile, stack, LIFO (Last In First Out)

- Algorithmes correspondants aux 4 opérations

```
Fonction pilevide(Pile p, entier cpt): booléen
Début
    retourne (cpt==0)
Fin
```

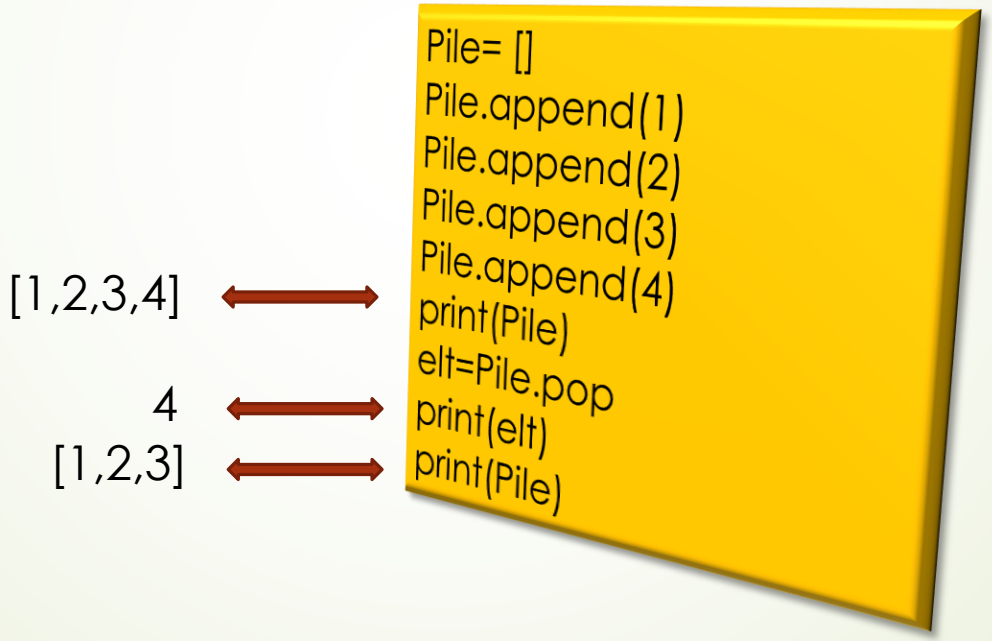
```
Fonction pilepleine(Pile p, entier cpt, entier taillemax): booléen
Début
    Retourne (cpt==taillemax)
Fin
```

Pile, stack, LIFO (Last In First Out)

- Son utilisation la plus importante : gérer l'appel de sous-programmes (procédures, fonctions, méthodes)
- La fonction « Annuler la frappe » (en anglais Undo) mémorise les modifications apportées au texte dans une pile.
- Parseur d'expressions XML, des pages web
- Un algorithme de recherche en profondeur dans un graphe utilise une pile pour mémoriser les nœuds visités.
- Les algorithmes récursifs utilisent implicitement une pile d'appels

Pile, stack, LIFO (Last In First Out)

- En Python on implémente une pile avec une liste
  - Empiler: .append
  - Depiler: .pop
  - pilevide: len(pile)==0
  - Création: pile= []

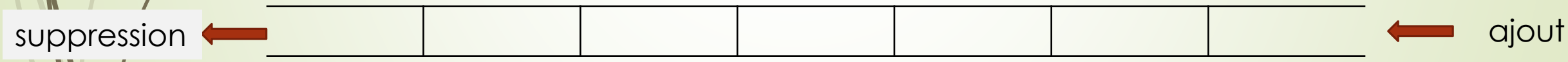


File (d'attente), queue, FIFO (First In First Out)

Sortie de la file

entrée dans la file

C'est celui qui attend depuis le plus longtemps qui sera le premier servi



File (d'attente), queue, FIFO (First In First Out)

Enfiler: .append()
Défiler: .pop(0)
Filevide:
len(file)==0
Création : pile=[]

[0,1,2,3] ⟺
0 ⟺
[1,2,3] ⟺

```
File=[]
File.append(0)
File.append(1)
File.append(2)
File.append(3)
print(File)
Elt=File.pop(0)
print(Elt)
print(File)
```