UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en Python

Cours 5. Listes, tuples et gestion mémoire

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 1 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Examen de mi-semestre

- Examen le jeudi 21 mars de 15h30 à 17h30
  - Sur les chapitres 1 à 5
  - On vous tiendra au courant du lieu
- En cas d'incompatibilité avec une autre UE.
  - Débrouillez-vous avec l'autre enseignant :)
  - Me contacter par mail
- Il n'y aura pas de cours de Python la semaine du 20 Mars.
  - :'(
  - Ni CM, ni TP, ni TD

Sommaire

- Une séquence est une suite finie de valeurs indexées.
  - On connaît déjà le type `str` (chaîne de caractères)
  - Il y a aussi les tuples et les listes.

SHELL
```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

- Une séquence peut contenir des éléments de types distincts :

SHELL
```
>>> t = (2.0 , 'Salut', math.cos)# tuple de 3 éléments
>>> [1, 'B', t , [3,4,5]]        # liste de 4 éléments
[1, 'B', (2.0, 'Salut', <built-in function cos>), [3, 4, 5]]
```

- Les tuples sont une généralisation des couples.
  - couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner
qu'une valeur,
mais peut retourner un tuple.

SCRIPT

```
def division(a,b):
    q = 0
    r = a
    while r >= b:
        r = r - b
        q = q + 1
     # q=a//b et r=a%b
    return (q, r)
```

SHELL

```
>>> t = division(37,7)
>>> t # 37 = 5*7 + 2
(5, 2)
>>> t + (3,4) # concaténation
(5, 2, 3, 4)
>>> t[0]=17 # les tuples ne sont pas modifiables
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

- Les listes sont une généralisation des tuples.
- Exemple :
  - On souhaite conserver les prénoms des membres d'un club
  - On peut utiliser les listes pour les stocker.

SHELL
```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']
>>> club = club + ['Etienne', 'Fatou']
>>> club
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

- Zut alors ! On a oublié l'accent sur Étienne ! ! !

SHELL
```
>>> club[4]='Étienne'
>>> club
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Étienne', 'Fatou']
```

- Les listes sont mutables.

Sommaire

- Construction en extension
  - tuples : les éléments entre parenthèses séparés par des virgules
  - liste : les éléments entre crochets séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	`()`	`(2,)`	`(3,4,1,(1,3),-6)`
Liste	`[]`	`[2]`	`[3,4,1,(1,3),-6]`
Chaîne	`''`	`'@'`	`'Salut'`

  - Un 1-uplet est possible mais peu utile : attention à la syntaxe `(2,)`.

- La fonction `len(S)` donne la longueur (nombre d'éléments de S)

SHELL
```
>>> len('')
0
>>> len((2,))
1
>>> len([0,1,2])
3
```

Opérations sur les séquences

- On peut concaténer deux séquences avec l'opérateur +

SHELL
```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

- En particulier, si S1 et S2 sont deux séquences on a

```
len(S1+S2) == len(S1) + len(S2)
```

- On peut multiplier une séquence par un entier :

SHELL
```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> 4 * [ 3,4,5]
[3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5]
>>> 'HA '* 10  + 5*'!'
'HA HA HA HA HA HA HA HA HA HA !!!!!'
```

- Les opérations sont communes à ces trois types de séquence.

- L'accès est commun aux chaînes, tuples et listes.

SHELL
```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1,True,'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL
```
>>> len(l)
3
>>> l[0] #Mon 1er est un chiffre
6
>>> l[1] #Mon 2nd est un booléen
True
>>> l[2] #Mon 3ème est une chaîne
'Yeux'
>>> l    #Mon tout est orange
[6, True, 'Yeux']
```

- Comme pour les chaînes, on peut utiliser des indices négatifs.

SHELL
```
>>> l[-1] , t[-2] , c[-3] # Sans parenthèses
('Yeux', True, 'n')
>>> t[666] # Horreur, une erreur
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: tuple index out of range
```

- Pour parcourir une séquence, on peut utiliser des boucles.
  - en itérant sur les indices de 0 à `len(Seq)-1`
  - en itérant directement sur les éléments de la séquence.

SCRIPT
```
def affiche(Seq):
    n=len(Seq)
    for i in range(n):
        print(Seq[i],end=' -> ')
    print('')
```

SCRIPT
```
def affiche(Seq):
    for e in Seq:
        print(e,end=' -> ')
    print('')
```

SHELL
```
>>> affiche(c)
S -> a -> l -> u -> t -> o -> n ->   -> ! ->
>>> affiche(t)
1 -> True -> badour ->
>>> affiche(l)
6 -> True -> Yeux ->
```

- C'est le même programme pour les trois types de séquences !

Écrire une fonction `index`

- On souhaite connaître l'indice d'un élément d'une séquence
  - On retourne le premier indice qui convient
  - Si aucun indice ne convient, on renvoie -1

SCRIPT

```
def index(Seq,x):
    for i in range(len(Seq)):
        if Seq[i]==x:
            return i
    # On sort de la boucle si on n'a pas trouvé x
    return -1
```

- Et ça marche avec tous les types de séquence !

SHELL

```
>>> print(c,t,l,sep='    ')
Saluton !    (1, True, 'badour')    [6, True, 'Yeux']
>>> index(c,'o')
5
>>> index(t,False)
-1
>>> index(l,True)
1
```

- Une méthode similaire existe déjà.
  - Mais il faut savoir l'écrire soit même !

SHELL
```
>>> liste = [4, -2, False, 'Coucou',-2, (3,5)]
>>> liste.index('Coucou')
3
```

  - En cas de plusieurs index valides, c'est le premier qui est renvoyé.
  - On peut donner un indice de départ
  - Si aucun n'est valide, le programme se termine par un erreur.

SHELL
```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  5 is not in list
>>> (10,20,30).index(30)  # marche avec tuples et chaînes
2
>>> 'abcdef'.index('bcd') # marche avec des sous-chaînes
1
```

- On peut déstructurer un tuple ou une liste.
  - nécessite de connaître la taille en écrivant le programme.
  - permet de définir plusieurs variables d'un coup

SHELL
```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL
```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

- Il faut la même structure des deux côtés.

SHELL
```
>>> liste=[1,2,3]
>>> [a,b] = liste
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  too many values to unpack (expected 2)
```

- Comme pour les chaînes, on peut extraire des tranches.
  - On accède à un élément d'une séquence avec la notation `[i]`
  - On accède à une suite d'éléments avec la notation `[i:j]`
  - On accède à une suite d'éléments avec la notation `[i:j:p]`

SHELL
```
>>> c = 'Salut à toi'
>>> l = ['Zéro','Un','Deux','Trois','Quatre','Cinq']
>>> t = (0,1,2,3,4,5)
```

SHELL
```
>>> c[6]
'à'
>>> l[1:4]
['Un', 'Deux', 'Trois']
>>> t[1:6:2]
(1, 3, 5)
```

SHELL
```
>>> extraire(c,6,7,1)
'à'
>>> extraire(l,1,4,1)
['Un', 'Deux', 'Trois']
>>> extraire(t,1,6,2)
(1, 3, 5)
```

- Exercice : Écrire la fonction extraire.

- Une construction par compréhension (de liste) consiste à imiter la notation mathématique

$$\{ f(x) \text{ tels que } x \in E \text{ et } P(x) \}$$

SHELL
```
>>> [x * x for x in range(6)]
[0, 1, 4, 9, 16, 25]
>>> [c for c in 'azerty']
['a', 'z', 'e', 'r', 't', 'y']
```

- Une expression `if` suivant le `for` permet de filtrer les éléments.

SHELL
```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]
[36, 64, 100, 144, 196, 256, 324]
```

- Pour les tuples la syntaxe est :

SHELL
```
>>> tuple(x*x for x in range(20) if x%2==0 and x*x>20)
(36, 64, 100, 144, 196, 256, 324)
```

- Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]
[0, 1, 4, 9, 16, 25]
```
SHELL

```
>>> [c for c in 'azerty']
['a', 'z', 'e', 'r', 't', 'y']
```
SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]
[36, 64, 100, 144, 196, 256, 324]
```
SHELL

- Même question avec une double boucle `for`

```
>>> L = [ (1,2), (3,4,5), (6,) ]
>>> [x for tuple in L for x in tuple]
[1, 2, 3, 4, 5, 6]
```
SHELL

- On peut convertir une chaîne en tuple ou en liste.

SHELL
```
>>> list('Saucisson')
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']
>>> tuple('Découpage')
('D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

- On peut convertir les listes en tuples et vice-versa

SHELL
```
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>> tuple([1,2,3,4])
(1, 2, 3, 4)
```

- Exercice : écrire les deux fonctions `tuple` et `list`.

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- On représente souvent les matrices comme une liste de listes

```
[[a,b,c],[d,e,f],[g,h,i]]
```

SHELL

```
>>> M = [[0,1,2],[10,11,12],[20,21,22]]
>>> M[1]
[10, 11, 12]
>>> M[1][2]
12
```

- Exercice 1 : Trouver le maximum d'une matrice

- Exercice 2 : Ajouter case par case deux matrices

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 50 & 60 \\ 70 & 80 \end{pmatrix} = \begin{pmatrix} 51 & 62 \\ 73 & 84 \end{pmatrix}$$

Sommaire

- On pose `liste=[11,22,33]` ; `s='Chaine'`, et `t = ('t','U','p','l','e')`
- On peut modifier les listes.

SHELL
```
>>> liste[1] = 'Choucroute'
>>> liste
[11, 'Choucroute', 33]
```

- On ne peut pas modifier les tuples et les chaînes

SHELL
```
>>> s[3]='î'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'str' object does not support item assignment
>>> t[1]='u'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
```

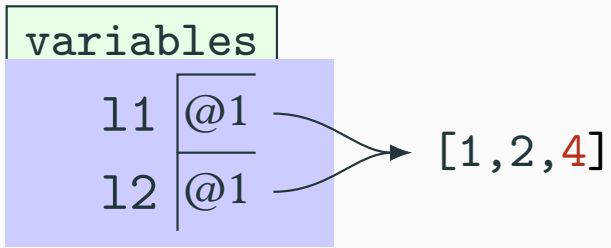
- On peut cependant réaffecter une chaîne et un tuple

SHELL
```
>>> s='Chaîne'
>>> t=('t','u','p','l','e')
```

SHELL
```
>>> c1 = 'Chaîne vachement très très longue'
>>> c2=c1
```

- La chaîne `'Chaîne … longue'` est stockée une fois en mémoire.
  - On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - l'affection c2=c1 est instantanée.

SHELL
```
>>> l1=[1,2,3]
>>> l2=l1
>>> l1[2]=4
>>> l1     # l1 modifié : logique…
[1, 2, 4]
>>> l2     # Surprise !!!
[1, 2, 4]
```



Ajout d'un élément

SHELL
```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL
```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

- Le premier programme
  - recrée une liste similaire à `L` (peut être long si `L` est grand)
  - ajoute 3 à la fin de la nouvelle liste.
- La méthode `append`
  - modifie la liste `L` directement.

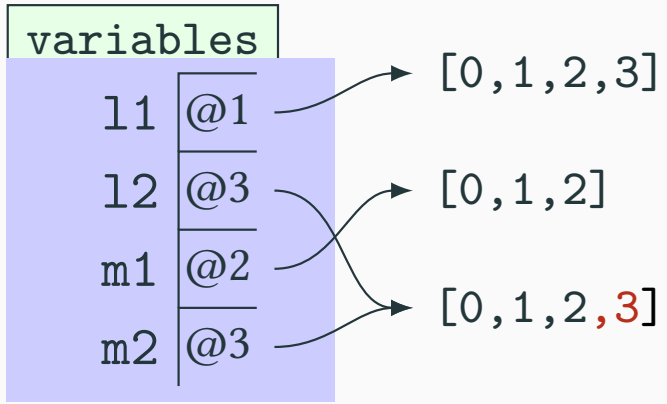
SHELL
```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas recopiée
>>> m2=l2 # La liste n'est pas recopiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>> l2.append(3) # on modifie la liste
>>> print(l2,m2)
[0, 1, 2, 3] [0, 1, 2, 3]
```

Ajout d'un élément : explication

- Python ne stocke pas des listes dans des variables (manque de place)
- Python stocke les adresses en mémoire des listes.

SCRIPT

```
l1=[0,1,2]
l2=[0,1,2]
m1=l1
m2=l2
l1=l1+[3]
l2.append(3)
```



- `l1` contient l'adresse @2 qui pointe vers une liste `[0,1,2]`
- Remarquons que :
  - `l1`≠`l2` : adresse différente
  - `l1`=`m1` : même adresse
- `l1=...` on modifie `l1` (on remplace @2 par @1). Maintenant `l1`≠`m1`
- On ne modifie pas `l2` (toujours égal à @3). On a toujours `l2`=`m2`

Exemple : créer la liste des diviseurs

- La fonction `diviseurs(n)` va retourner la liste des diviseurs de n.
  - uniquement ceux non triviaux (distincts de 1 et `n`)
  - on impose à `n` d'être un entier strictement positif.

- Liste ou tuple ?
  - On peut efficacement ajouter des éléments à une liste : `append`
  - Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - On ne connaît pas à l'avance le nombre d'éléments du résultat
  - On va donc utiliser une liste plutôt qu'un tuple

SCRIPT

```
def diviseurs(n):
    res = []
    for d in range(2, n):
        if n % d == 0 : # Si d divise n
            res.append(d)
    return res
```

SHELL

```
>>> d = diviseurs(500)
>>> (len(d) , d)
(10, [2, 4, 5, 10, 20, 25, 50, 100, 125, 250])
```

Méthodes usuelles sur les listes

`https://docs.python.org/fr/3.7/tutorial/datastructures.html`

`list.append(x)`	Ajoute un élément à la fin de la liste.
`list.extend(L)`	Étend la liste en y ajoutant tous les éléments de l'itérable.
`list.insert(i,x)`	Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer.
`list.remove(x)`	Supprime de la liste le premier élément dont la valeur est égale à x. Erreur s'il n'existe aucun élément avec cette valeur.
`list.pop(i)`	Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, `a.pop()` enlève et renvoie le dernier élément de la liste.
`list.index(x)`	Renvoie la position du premier élément de la liste dont la valeur égale x. Erreur si aucun élément n'est trouvé.
`list.count(x)`	Renvoie le nombre d'éléments ayant la valeur x dans la liste.
`list.sort()`	Ordonne les éléments dans la liste, en place.
`list.reverse()`	Inverse l'ordre des éléments dans la liste, en place.

- Exercices : à réécrire chez soi :)

Sommaire

- Calculons la liste des chiffres d'une chaîne s.
  - Nous utilisons la méthode `isdigit()` de la classe str.
  - Vous devez être capable d'écrire `isdigit` ! (*cf.* TP 3)

SHELL
```
>>> '456'.isdigit()
True
```

SHELL
```
>>> '45a6'.isdigit()
False
```

- Méthode classique avec boucle `for`.

SCRIPT
```
def chiffres(chaîne):
    liste = []
    for c in chaîne:
        if c.idigit():
            liste.append(c)
    return liste
```

SHELL
```
>>> chiffres('01ivier E5T 2314 fois 5YMP4')
['0', '1', '5', '2', '3', '4', '5', '4']
```

- Méthode Pythonesque : compréhension de liste.

SCRIPT

```
def chiffres(s):
    return [c for c in s if c.isdigit()]
```

- Notez l'ordre des instructions :
  - D'abord le `for` puis le `if`
  - Comme dans le programme précédent.

- Ne marche pas (forcément) avec les autres langages.

- Durant cette UE privilégiez la méthode classique.

- Principes : (Déjà vu dans un cours précédent : Cours 2)
  - on traite les nombres pairs (divisibles par 2) à part,
  - `d` divise `n` si et seulement si `n%d == 0`,
  - `n` non premier admet nécessairement un diviseur $d \leq \sqrt{n}$

SCRIPT

```
def ppdiv(n):
    if n%2==0:
        return 2
    racine = int(sqrt(n))
    for d in range(3,racine+1,2):
        if n%d==0:
            return d
    return n
```

SHELL

```
>>> ppdiv(2003) # premier
2003
>>> ppdiv(1003) # non premier
17
>>> [(c,ppdiv(c)) for c in range(15,20)]
[(15, 3), (16, 2), (17, 17), (18, 2), (19, 19)]
```

- Il est facile de tester si un nombre est premier en utilisant `ppdiv`...

SCRIPT
```
def estPremier(n):
    if n<2:
        return False
    else:
        return ppdiv(n)==n
```

- ... puis de construire la liste des nombres premiers jusqu'à n :

SCRIPT
```
def premiers(n):
    liste = []
    for p in range(n + 1):
        if estPremier(p):
            liste.append(p)
    return liste
```

SHELL
```
>>> premiers(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

- Exercice : écrire cette fonction en une ligne (par compréhension)

- Python possède deux primitives pour trier une liste `L` de nombres.
  - la fonction `sorted(L)` qui construit une nouvelle copie de `L`
  - La méthode `L.sort()` qui trie la liste `L` en place (sans créer de nouvelle liste, en modifiant le contenu de `L`).

SHELL
```
>>> L1 = [3, 1, 6, 5, 2]
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>> L1
[3, 1, 6, 5, 2]
>>> L1 == sorted(L1)
False
```

SHELL
```
>>> L2 = [3, 1, 6, 5, 2]
>>> L2.sort()
>>> L2
[1, 2, 3, 5, 6]
>>> L2 == sorted(L2)
True
```

- Le temps de calcul de cet algorithme pour trier une liste à n éléments est proportionnel à $n \cdot log(n)$. Petite expérience :

SCRIPT

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()        # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SHELL

```
>>> chrono(200000)
Temps pour 200000 24 ms
>>> chrono(400000)
Temps pour 400000 51 ms
```

- En doublant `n`, on fait un peu plus que doubler le temps de calcul
  - ce qui est cohérent avec « $n \cdot log(n)$ »

- Il existe de nombreux algorithmes de tri.
  - Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - les meilleurs sont en $n \cdot log(n)$
  - Le tri par sélection est proportionnel à n^2
- Concrètement

[5,6,7,2,4]

[5,6,7,2,4]
[2,6,7,5,4]
[2, 6,7,5,4]
[2, 4,7,5,6]
[2,4, 7,5,6]
[2,4, 5,7,6]
[2,4,5, 7,6]
[2,4,5, 6,7]
[2,4,5,6, 7]

- `imin` trouve le minimum à partir de `i` et renvoie son indice

SCRIPT
```
def imin(L, i):
    m = i
    for j in range(i + 1, len(L)): #  i+1 … len(L)-1
        if L[j] < L[m]:
            m = j
    return m
```

- L'algorithme de tri par sélection

SCRIPT
```
def tri(L):
    n = len(L)
    for i in range(n-1):
        m = imin(L,i)  # minimum de la fin de liste
        (L[i], L[m]) = (L[m], L[i]) # échange
        print('L=', L) # pour espionner la boucle
```

- À la fin de chaque étape les `i` premiers éléments sont triés.
- Par super efficace, mais ça marche !

SCRIPT

```
def chrono2(f,n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time() ; f(L) ; t = time() - t
    print(f.__name__,':',int(t*1000),'ms')
```

SHELL

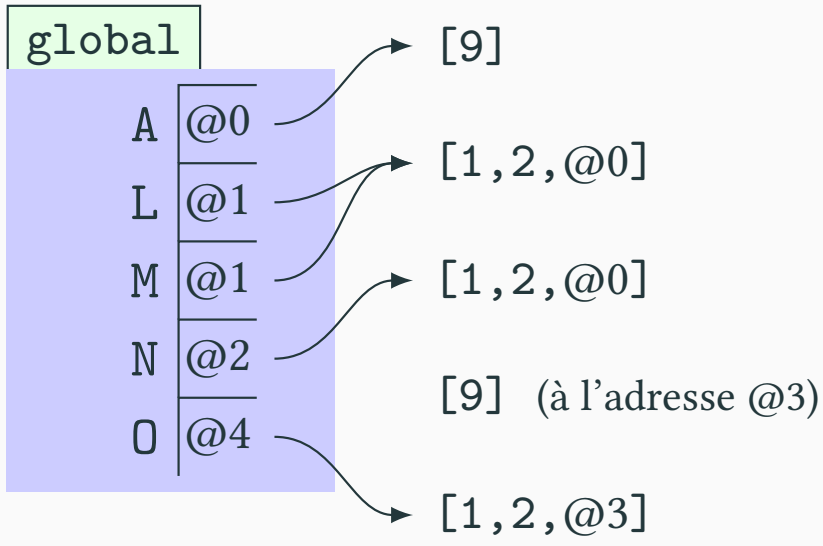
```
>>> L = [4,2,7,11,2,1]
>>> tri(L)
L = [1, 2, 7, 11, 2, 4]
L = [1, 2, 7, 11, 2, 4]
L = [1, 2, 2, 11, 7, 4]
L = [1, 2, 2, 4, 7, 11]
L = [1, 2, 2, 4, 7, 11]
>>> chrono2(sorted,10000)
sorted : 2 ms
>>> chrono2(tri,10000) # Après avoir enlevé les prints
tri : 1818 ms
```

Sommaire

- Une simple affectation (`L=M`) copie simplement le pointeur.
  - Avantage : rapide
  - Inconvénient : dangereux et ne permet pas de copier.
- Il existe une méthode `M.copy()` pour copier la liste.
- Il existe aussi une méthode `M.deepcopy()` pour la copier en profondeur.

SCRIPT

```
from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant
```



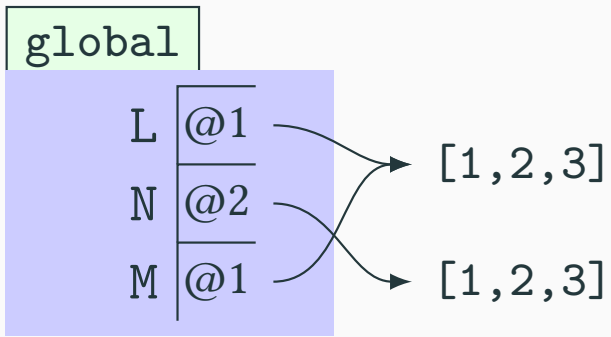
- À savoir écrire vous même !

Être ou ne pas être égal...

- Que signifie `L` et `M` sont la même liste ?
- À ne pas confondre :
  - L'égalité testée avec `==` : les listes ont le même contenu.
  - L'identité testée avec `is` : les listes pointent au même endroit mémoire.

SHELL

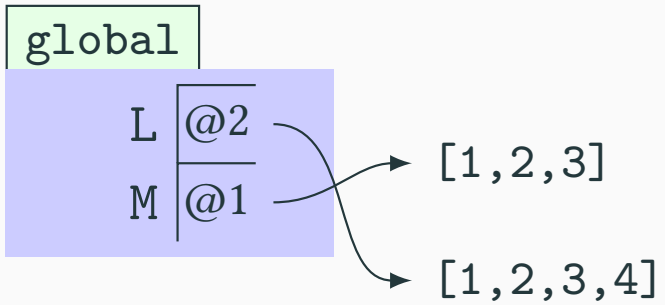
```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
True True
>>> print(M is N , M==N) # Non identiques mais égales
False True
```



- Exercice : écrire une fonction `copy(L)` qui retourne une liste égale mais distincte de L.

SCRIPT

```
L = [1,2]
M = L
L.append(3)
L = L + [4]
```

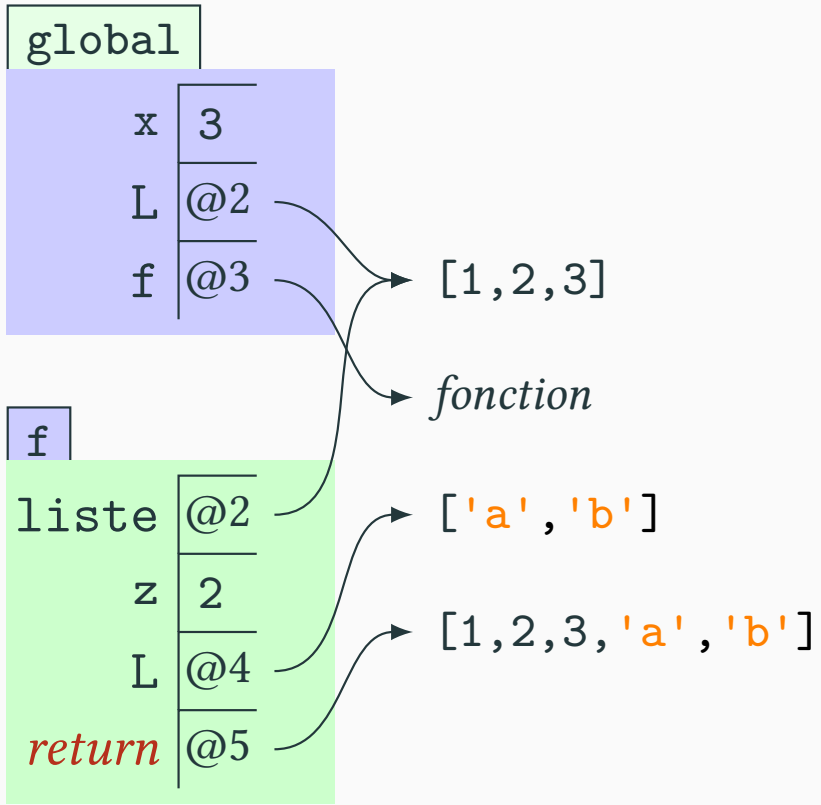


- L'appel de `L.append(3)` prolonge la liste
  - Ne modifie pas L (toujours égale à @1), seulement le contenu de la liste.
  - est très rapide
  - a un effet global !
- Le calcul de `L+[4]` crée une nouvelle liste
  - Peut être très long si L contient beaucoup d'éléments
  - Sécurisé mais souvent inutile
  - Crée un nouvel objet en mémoire

- Pour simplifier, il y a trois endroits où Python stocke des informations.
  - La mémoire des variables globales.
  - La mémoire des variables locales communément appelée pile
  - La mémoire des objets (liste, fonction, etc) communément appelée tas
  - Une variable peut contenir un lien vers un objet du tas.

SCRIPT

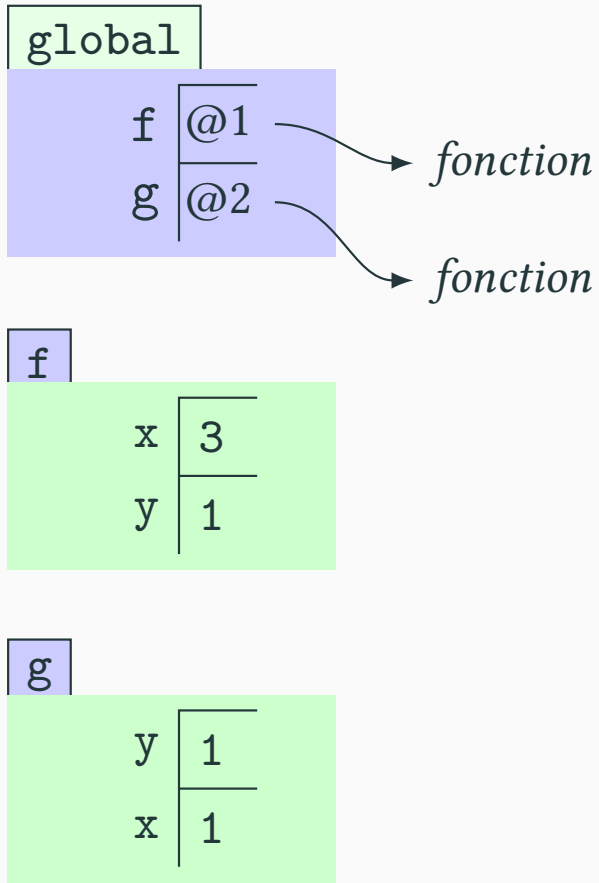
```
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```



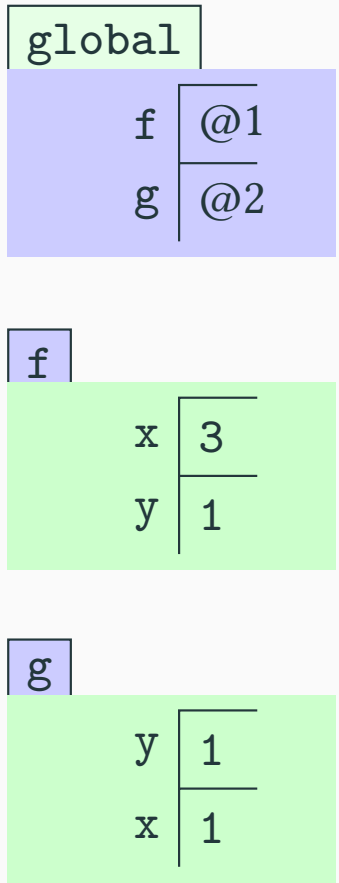
SCRIPT

```
def f(x):
    y = 1
    g(y)
    print(x)

def g(y):
    x = y

f(3) # affiche 3
```

- On définit `f`
- On définit `g`
- On appelle `f(3)`
- On appelle `g(y)` dans `f`
- `g` se termine, on libère sa mémoire
- `f` se termine, on libère sa mémoire



- Chaque appel de fonction ajoute un cadre sur la pile.
  - Les cadres s'empilant, ils forment une pile.
  - Il ne dure que le temps de l'appel.
  - Ensuite, il disparaît avec ses variables locales.
- Chaque appel de fonction a son propre espace de nom.
  - Le `x` de `f` n'est pas le même que celui de `g`
  - Cela permet de ne pas s'embrouiller
- Si on appelle plusieurs fois une fonction, il y aura un cadre par appel.

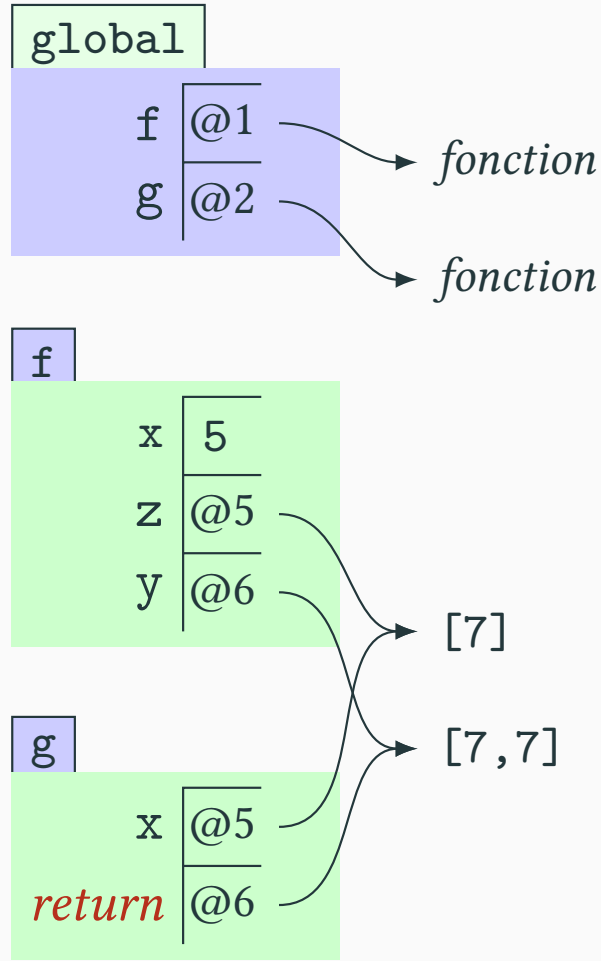


Partage de variables locales entre fonctions

- Il est impossible de partager des variables locales entre fonctions
- Mais on peut se transmettre des valeurs (du tas) entre fonctions :
  - par passage d'argument `g(z)`
  - par valeur de retour `return x+x`

SCRIPT

```
def f(x):
    z=[x+2]
    y=g(z)

def g(x):
    return x+x

f(5)
```

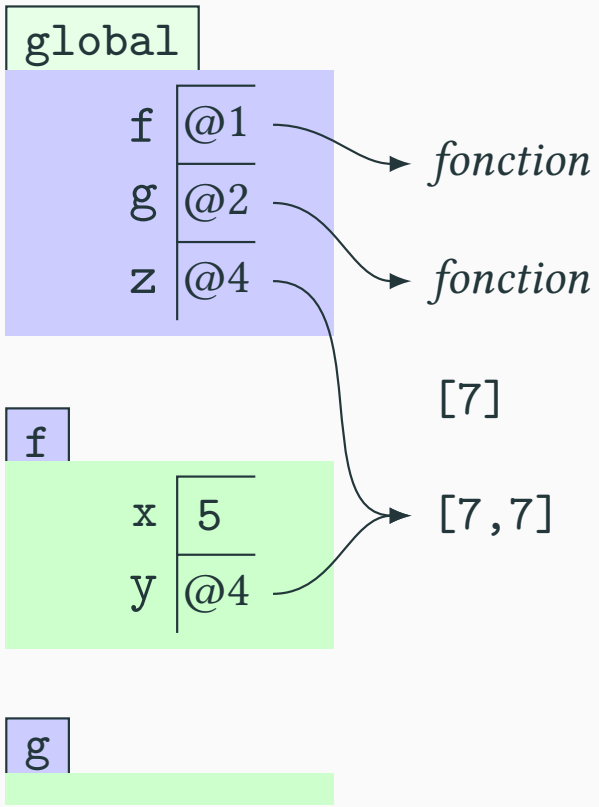


C'est la manière propre

- On peut aussi utiliser des variables globales (cours 4)
- Ici, `f` et `g` vont modifier la variable globale `z`.

SCRIPT

```
def f(x):
    global z
    z=[x+2]
    g()
    y=z

def g():
    global z
    z=z+z

f(5)
```



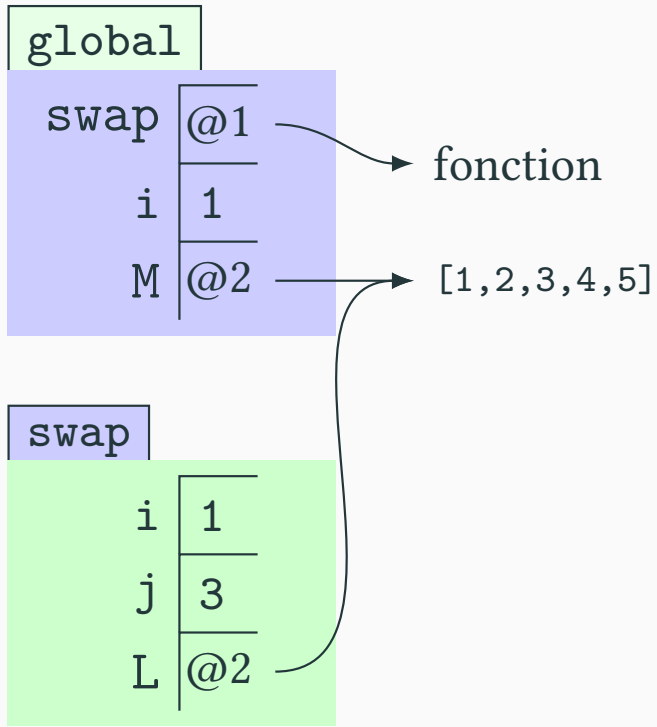
C'est la manière (très !) sale

#balancetongoret

- Attention : le contenu d'une liste est toujours modifiable globalement.
  - La variable `L` contient un lien vers la liste `[1,2,3]`
  - Ce lien n'est pas modifiable (par défaut) par une fonction (variable locale).
  - Le contenu pointé par le lien est toujours modifiable.

SCRIPT

```
def swap(i,j,L):
    (L[i], L[j]) = (L[j], L[i])
    i = i + 1

i = 1
M = [1, 2, 3, 4, 5]

swap(i,3,M)
print(M) # affiche [1, 4, 3, 2, 5]
print(i) # affiche 1
```



- Pas de `global M` dans la fonction `swap`
- Donc M n'est pas modifié (pointe toujours vers la même liste)
- Mais le contenu de cette liste a changé !

SCRIPT

```
class Point:

    def __init__(self,x,y):
        self.x = x
        self.y = y


def translation(q): # q n'est pas une variable globale
    q.x = q.x + 20
    q.y = q.y + 30
```

SHELL

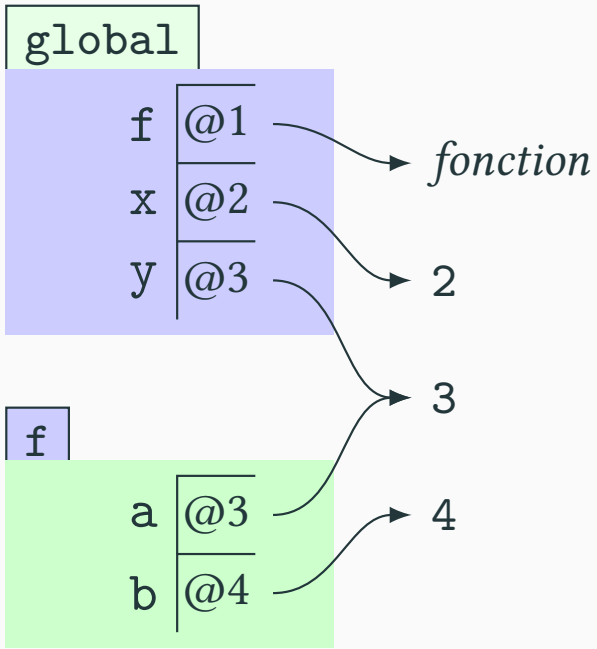
```
>>> p = Point(0,0)
>>> (p.x,p.y)
(0, 0)
>>> translation(p)
>>> (p.x,p.y)
(20, 30)
```

- Techniquement, on ne modifie pas la variable p (contenant le pointeur)...
  - ...seulement son contenu.
- Attention danger !

- En Python tout est objet
- En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

SCRIPT

```
def f(a):
    b=a+1

x=2
y=3
f(y)
```



- Cela signifie-t-il que `f` peut modifier `y` à travers `a` ?

- La réponse est non !
  - `f` ne peut pas modifier le pointeur dans y (pas de `global`)
  - `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- La majorité des types sont heureusement immuables.
  - `int`, `float`, `str`, `tuple`, `Nonetype`
  - Il serait gênant de pouvoir modifier la valeur du nombre 3 !

SHELL
```
>>> x=3
>>> x=4
>>> L=[1]
>>> L.append(3)
```

  - je modifie `x` mais pas l'entier `3`
  - Je modifie la liste `[1]` (qui devient `[1,3]`) mais pas `L` !
- Seuls les objets plus complexes (listes, ou objets créés par l'utilisateur) sont mutables.
  - source de bogues !

Merci pour votre attention
Questions

?

Cours 5 — Listes, tuples et gestion mémoire