Bases de l'informatique 1

Cours 6. Listes, gestion mémoire et exceptions

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

Partiel de mi-semestre

- Examen le mercredi 13 novembre de 13h15 à 15h15
  - Sur les chapitres 0 à 4
  - 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 cette semaine là.
  - :'(
  - Ni CM, ni TP, ni TD

Sommaire

- Techniquement, sous Python, les chaînes sont indexées :

de `-len(texte)` à `len(texte)-1`

Exemple : `texte='123 soleil'`

`texte[i]`	`'1'`	`'2'`	`'3'`	`' '`	`'s'`	`'o'`	`'l'`	`'e'`	`'i'`	`'l'`
`indice`	0	1	2	3	4	5	6	7	8	9
`indice`	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

SHELL

```
>>> texte='123 soleil'
>>> texte[-1]
'l'
>>> texte[-10]
'1'
>>> texte[-11]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError:  string index out of range
```

- En pratique `texte[-k]` est un raccourci pour `texte[len(texte)-k]`.

- 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):
    for i in range(len(seq)):
        print(seq[i],end=' -> ')
    print('')
```

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

SHELL
```
>>> affiche('Saluton !')
S -> a -> l -> u -> t -> o -> n ->   -> ! ->
>>> affiche((1,True,'badour'))
1 -> True -> badour ->
>>> affiche([6, True, 'Yeux'])
6 -> True -> Yeux ->
```

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

- On a par ordre d'expressivité décroissante :
  - La boucle `while` (la plus expressive)
  - La boucle `for i in range(len(L))`
  - La boucle `for x in L` (la moins expressive)
- En gagnant en lisibilité, on perd en expressivité.
- Exemple : parcourir un chaîne en miroir.

SCRIPT

```
def affiche_miroir(chaîne):
    n = len(chaîne)
    for i in range(n):
        print(chaîne[n-i-1], end="")
    print()
```

SHELL

```
>>> affiche_miroir("un rats repus")
super star nu
```

- 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

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
```

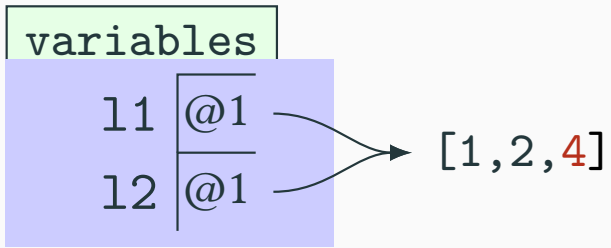
- On peut cependant réaffecter une chaîne et un tuple

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

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

- 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.

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



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.

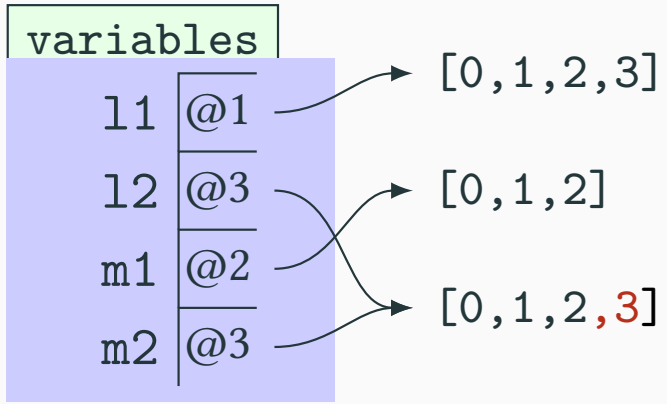
SHELL
```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas recopiée
>>> m2=l2 # La liste n'est pas recopiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>> l2.append(3) # on modifie la liste
>>> print(l2,m2)
[0, 1, 2, 3] [0, 1, 2, 3]
```

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 :)

- 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.isdigit():
            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.

Sommaire

- En cas d'erreur dans un programme, Python lève une exception.

SHELL
```
>>> L = [6, 4, 7, 2, 1, 8]
>>> L[6]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError:  list index out of range
>>> L.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  5 is not in list
>>> 2/0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError:  division by zero
```

- Il faut lire en détail les messages erreurs !
- La cause de l'erreur y est souvent clairement indiquée.

SCRIPT

```
1  def f(x):
2      print("Début du calcul")
3      a=g(x)
4      print("Fin du calcul")
5      return a+1
6
7  def g(x):
8      return 3%x
```

En cas d'erreur rencontrée :

- le programme s'interrompt immédiatement
- et affiche alors la trace d'exécution.

SHELL

```
>>> f(0) # f définie dans le fichier toto.py
Début du calcul
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "toto.py", line 3, in f
    a=g(x)
      ^^^^
  File "toto.py", line 8, in g
    return 3%x
           ~^~
ZeroDivisionError: integer modulo by zero
```

- Ligne 3 du fichier `toto.py` : `f` a rencontré une erreur : `a=g(x)`
- Ligne 8 du fichier `toto.py` : `g` a rencontré une erreur : `return 3%x`
- L'erreur est une division par zéro.

Éviter les exceptions

- Les exceptions peuvent être dues :
  - à une erreur de programmation.
  - à un utilisateur qui rentre de mauvaises valeurs.

SCRIPT
```
def record(chrono):
    if chrono < 9.58:
        return True
    else:
        return False
```

SHELL
```
>>> record('9s')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "sprint.py", line 2, in record
    if chrono < 9.58:
       ^^^^^^^^^^^^
TypeError:  '<' not supported between
instances of 'str' and 'float'
```

- On peut vérifier le type :

SCRIPT
```
def record2(chrono):
    if type(chrono) == float:
        return chrono < 9.58
    else:
        return False
```

SHELL
```
>>> record2(9.0)
True
>>> record2(9)
False
>>> type(9) == float
False
```

- Mais il est facile d'oublier des cas !

- Pour éviter que notre programme se termine,
  - On peut éviter les exceptions : cela peut rendre le code complexe et lourd
  - On peut les rattraper pour les traiter proprement.

SCRIPT
```
try:
    instruction # Bloc try
    instruction # Calcul qui peut
    instruction # lever des exceptions
except:
    instruction # Bloc except : Exécute
    instruction # seulement si une exception
    instruction # a été levée dans le Bloc 1
instruction # Suite du programme
```

- Dans un bloc `try`, on lance un code qui peut lever une exception.
- Si aucune exception n'est levée, on passe à la suite du programme.
- Si une exception est levée dans le bloc `try` :
  - on interrompt le bloc `try` à partir de l'exception
  - on exécute le bloc `except`
  - on passe alors à la suite du programme

SCRIPT

```
def record3(chrono):
    try:
        print('On essaie le calcul')
        résultat =  (chrono < 9.58)
        print('Tout c'est bien passé :)')
    except:
        print('Au secours une erreur !!!')
        résultat = False
    return résultat
```

- Si le test lève une exception, on considère que le record n'est pas battu.

SHELL

```
>>> record3(9.6)
On essaie le calcul
Tout c'est bien passé :)
False
>>> record3(9)
On essaie le calcul
Tout c'est bien passé :)
True
>>> record3('huit secondes et des poussières')
On essaie le calcul
Au secours une erreur !!!
False
```

Exemple : la fonction `index`

- On cherche à écrire une fonction `recherche(x,seq)` :
  - qui renvoie l'indice `x` dans la séquence `seq`.
  - qui renvoie `-1` si l'élèment n'est pas dans `seq`.
  - cf. cours 3 page 25 (`position`).

- Il existe une méthode `index`, mais elle lève une exception en cas d'échec.

SHELL
```
>>> ['a','b','c','b','c'].index('c')
2
>>> ['a','b','c','b','c'].index('d')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError:  'd' is not in list
```

- On va transformer cette « erreur » en `-1` en capturant l'exception.

Exemple : la fonction `recherche`

- On va utiliser la méthode `index`...
- ... et lorsqu'elle lance une exception...
  - on la rattrape au vol !
  - et on renvoie `-1`

SCRIPT

```
def recherche(x, seq):
    try:
        res = seq.index(x)
        # Pas d'erreur ? On continue.
        return res
    except:
        # Une erreur ? On exécute ce bloc.
        return -1
```

- Et cela fonctionne avec les chaînes, les listes et les tuples.

SHELL

```
>>> L=['a','b','c','b','c'] ; T=(1,2,3)
>>> recherche('c',L)
2
>>> recherche(4,T)
-1
```

- On veut calculer la moyenne d'une liste de note `moyenne(L)`.

SCRIPT
```
def moyenne(L):
    return  sum(L) / len(L) #  vous devez savoir écrire sum(…) !
```

- Si la liste est vide, on renvoie `"ABI"` *(absence injustifiée)*.
  - on va rattraper l'erreur de la division par zéro et seulement celle-ci

SCRIPT
```
def moyenne(L):
    try:
        return  sum(L) / len(L)
    except ZeroDivisionError:
        return "ABI"
```

SHELL
```
>>> moyenne([19,15,17])
17.0
>>> moyenne([])
'ABI'
>>> moyenne(12)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 3, in moyenne
TypeError:  'int' object is not iterable
```

Exemple : reconnaître un nombre

- La méthode `isdigit` ne fonctionne qu'avec les entiers.
- Astuce : on essaie de convertir une chaîne avec `float`
  - si ça marche, c'est que c'était possible…
  - sinon, c'est visiblement autre chose qu'un nombre flottant !

SHELL
```
>>> '1987'.isdigit()
True
>>> 'MCMLXXXVII'.isdigit()
False
>>> '6.55957'.isdigit()
False
```

SHELL
```
>>> float('6.55957')
6.55957
>>> float('MCMLXXXVII')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: could not convert string
to float: 'MCMLXXXVII'
```

SCRIPT
```
def est_nombre(s):
    try:
        float(s)
        return True
    except ValueError:
        return False
```

SHELL
```
>>> est_nombre('1987')
True
>>> est_nombre('6.55957')
True
>>> est_nombre('MCMLXXXVII')
False
```

Sommaire

Lever une exception : `assert`

- S'il y a trop peu d'erreurs dans votre code, vous pouvez en ajouter !
  - C'est une bonne pratique !
  - On ne cache pas d'erreurs mais on en informe l'utilisateur
  - Code plus court et lisible (on ne se préoccupe pas d'arguments non conformes)
- Conseil pour incarner la Zénitude !
  - L'explicite est mieux que l'implicite [...]
  - Les erreurs ne doivent jamais être passées sous silence, sauf de manière explicite
- `assert` : vérifie une propriété et lève une erreur en cas de non-respect.

SCRIPT

```
# assert
# On impose L non vide
def moyenne(L):
    assert L != []
    return sum(L)/len(L)
```

SHELL

```
>>> moyenne([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 2, in moyenne
AssertionError
```

- Rapide à mettre en place
- Permet d'être explicite sur ce qu'attend le programme.
- Message d'erreur un peu vague.

SHELL

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

SCRIPT

```
def f(L):
    """ f(L) renvoie un couple (n,b)
    n est la taille de L,  b vaut True si n est pair """
    pair=True
    for i in range(len(L)):
        pair=not(pair)
    return (i+1,pair)

assert f([2,3,4])   == (3,False) # 3 éléments : impair
assert f([2,3,4,6]) == (4,True)  # 4 éléments : pair
assert f([])        == (0,True)  # 0 élément  : pair
```

- si un test est valide, il ne se passe rien.
- sinon il y a une exception (ici l'erreur ne vient pas du `assert`)

SHELL

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "tp6.py", line 9, in <module>
    assert f([]) == (0,True) # 0 élément : pair
           ^^^^^

  File "tp6.py", line 5, in f
    return (i+1,pair)
            ^

UnboundLocalError:  cannot access local variable 'i' where it is not
associated with a value
```

- Méthode 1 : `assert`
  - Très pratique pour repérer les bogues qu'on ne devrait jamais rencontrer.
  - Permet de détecter les erreurs du développeur : utile pour les tests.
  - Mais messages peu clairs en production.
  - (même si on peut ajouter un message : `assert test, "Message"`)
- Méthode 2 : `raise`
  - Permet de préciser le type d'erreur
  - par exemple : `ValueError`, `IndexError`, `ZeroDivisionError`
  - Permet d'ajouter un message clair et explicatif.

SCRIPT

```
def moyenne(L):
    if L == []:
        raise ValueError('Liste vide : non mais allo quoi !')
    else:
        return sum(L) / len(L)
```

SHELL
```
>>> raise AssertionError('Il faut lire les consignes !!!')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AssertionError:  Il faut lire les consignes !!!
```

SHELL
```
>>>     raise ZeroDivisionError("Moyenne d'une liste vide impossible")
  File "<console>", line 1
    raise ZeroDivisionError("Moyenne d'une liste vide impossible")
IndentationError:  unexpected indent
```

SHELL
```
>>> raise ValueError('L argument doit être pair' + 1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError:  can only concatenate str (not "int") to str
```

SHELL
```
>>> raise IndexError("Je croyais la liste plus grande" + srt(1))
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError:  name 'srt' is not defined
```

- Vous pouvez utiliser `ValueError` et `TypeError`. Évitez les autres.
- On peut créer ses propres exceptions… mais nous ne verrons pas comment.

SCRIPT

```
def f():
    # déclenche une exception mais ne la rattrape pas
    2/0
    print('Ce message ne sera pas affiché.')

def g():
    f() # déclenche une exception
    print('Ce message non plus.')

def h():
    try:
        g()
        print('Ce message ne sera toujours pas affiché')
    except:
        print('Attrapée !')

h()
```

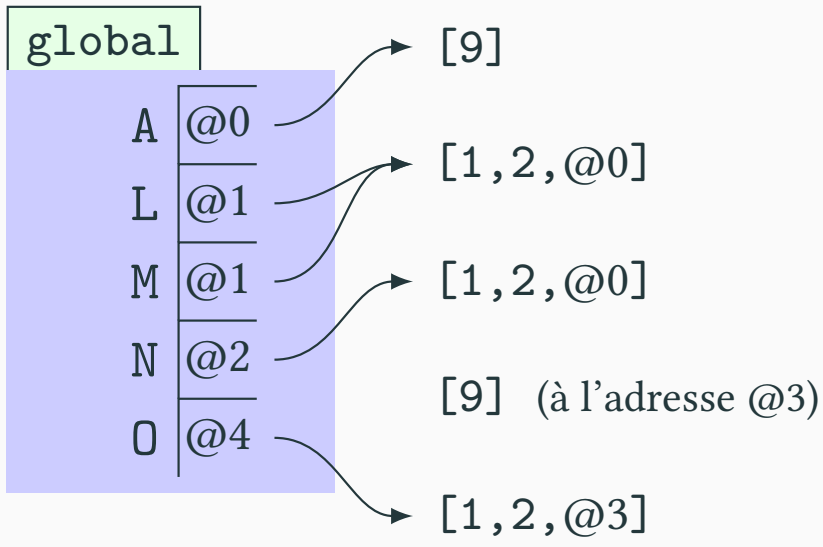
- Dans cet exemple l'exception interrompt la fonction `f()`,
- mais aussi la fonction `g()` qui appelle la fonction `f()`
- on passe directement de la ligne `2/0` à la ligne `print('Attrapée !')`

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

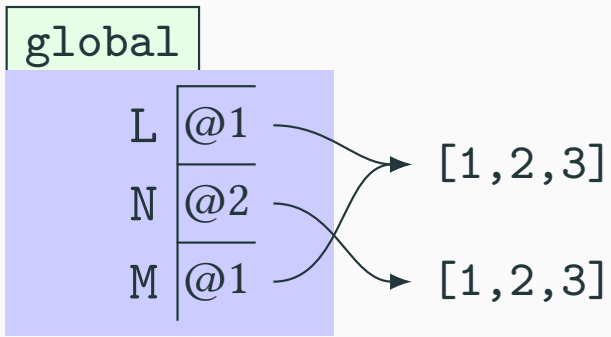
```
from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant
```



- À savoir écrire vous même !

- 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.

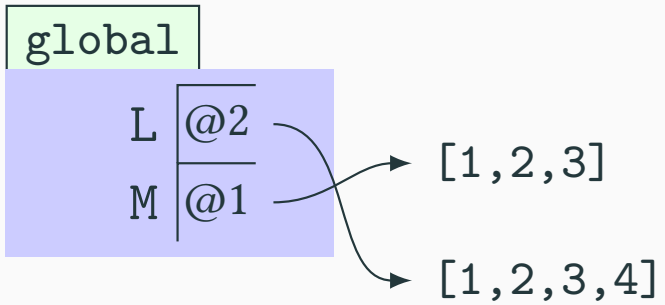
SHELL
```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
True True
>>> print(M is N , M==N) # Non identiques mais égales
False True
```



- 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]
```

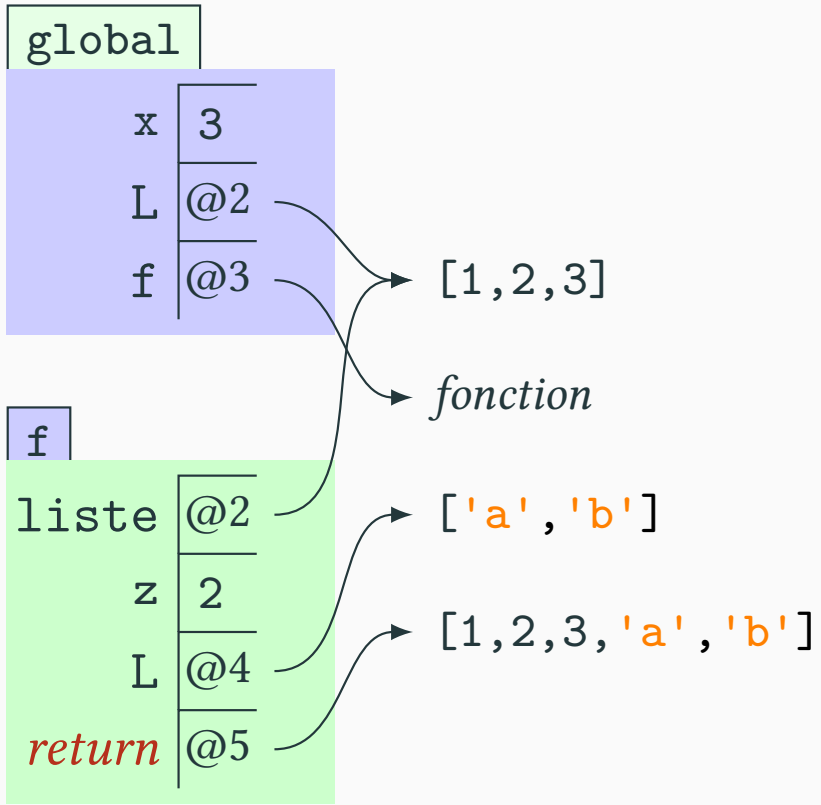


- L'appel de `L.append(3)` prolonge la liste
  - Ne modifie pas L (toujours égale à @1), seulement le contenu de la liste.
  - est très rapide
  - a un effet global !
- Le calcul de `L+[4]` crée une nouvelle liste
  - Peut être très long si L contient beaucoup d'éléments
  - Sécurisé mais souvent inutile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

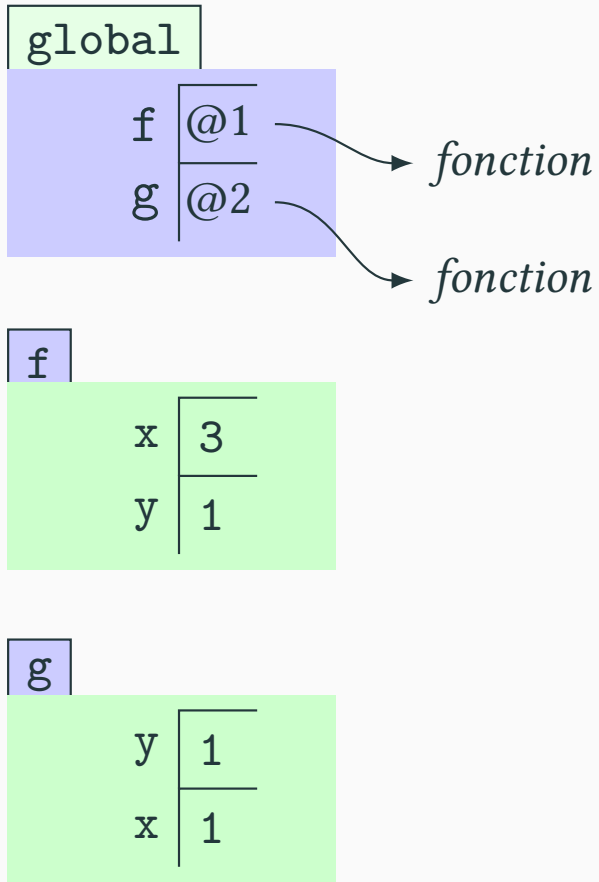
```
x=3
L=[1,2,3]

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

f(L)
```



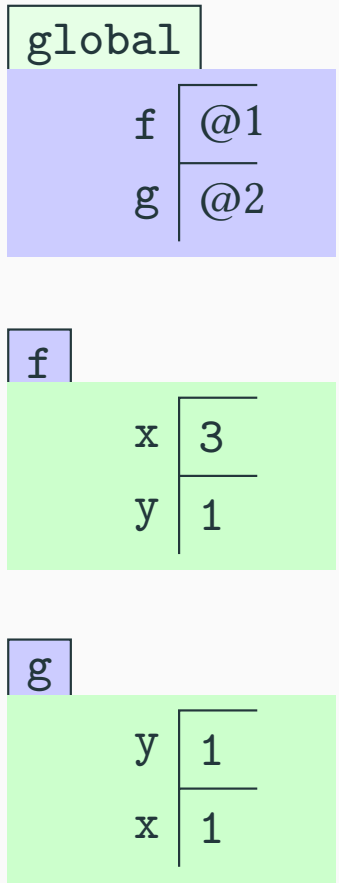
SCRIPT

```
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.

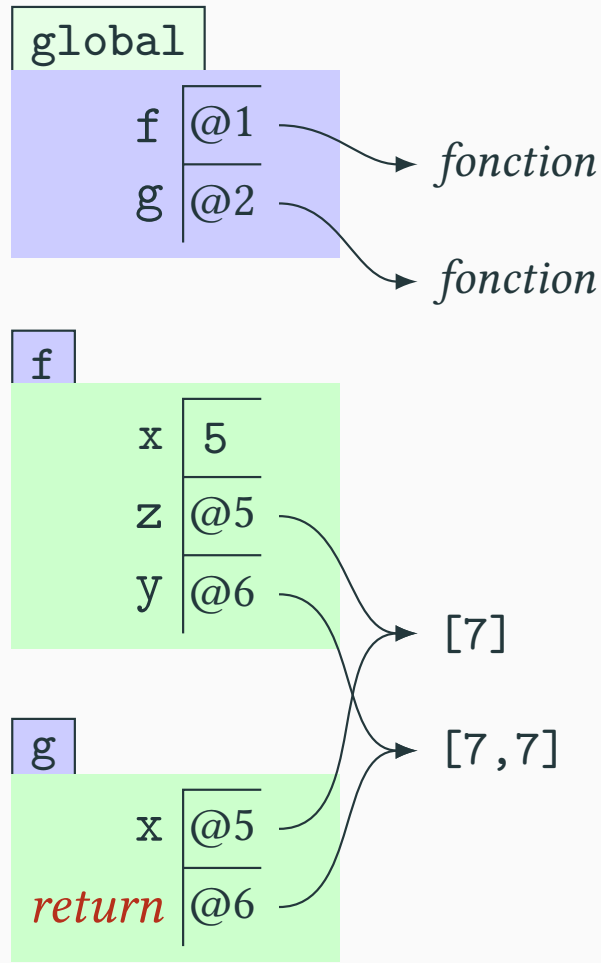


Partage de variables locales entre fonctions

- 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)
```



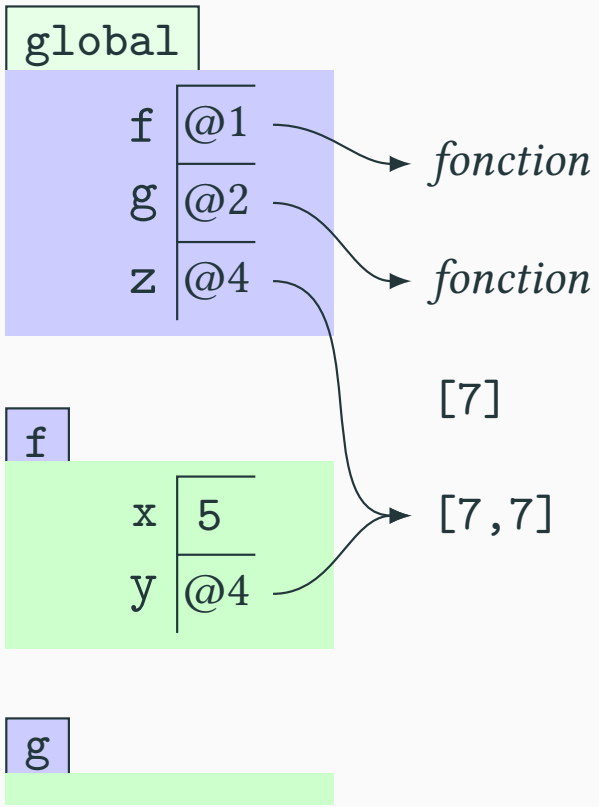
C'est la manière propre

Partage de variables globales entre fonctions

- 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)
```



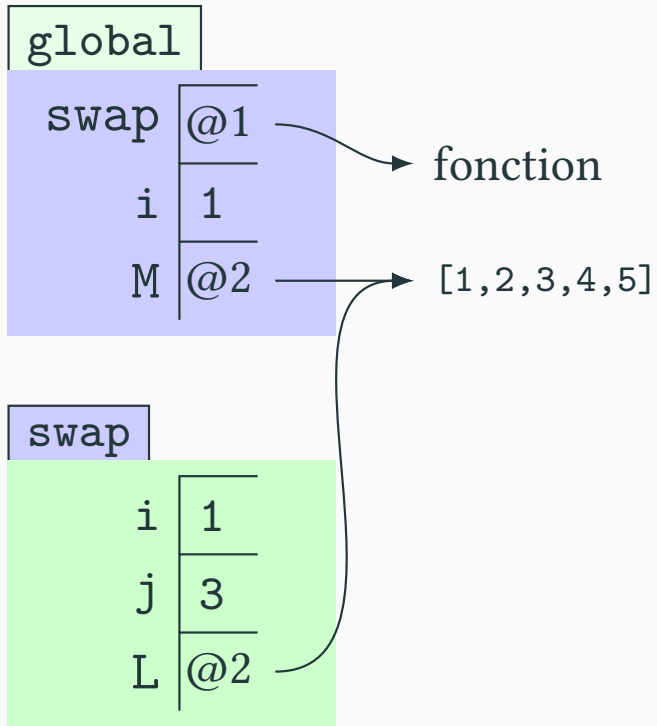
C'est la manière (très !) sale

#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

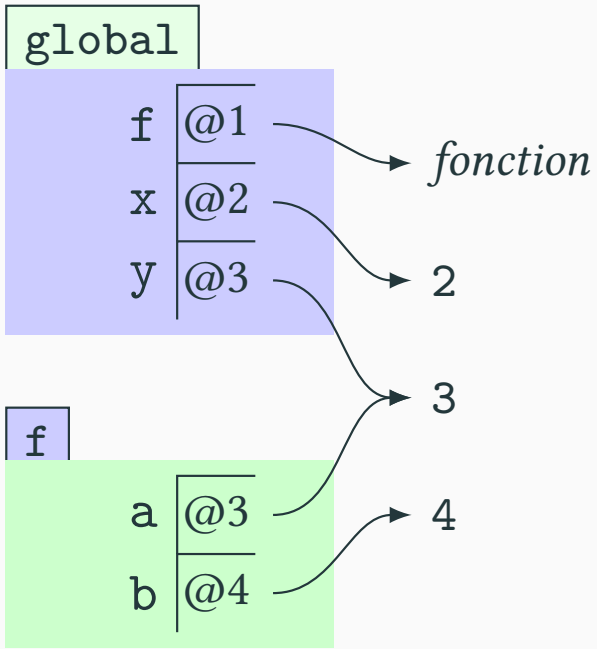
```
>>> p = Point(0,0)
>>> (p.x,p.y)
(0, 0)
>>> translation(p)
>>> (p.x,p.y)
(20, 30)
```

- 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 6 — Listes, gestion mémoire et exceptions

Licence