L3 Informatique - 2021-2022

Atelier 3 - Boucle

Paul-Antoine BISGAMBIGLIA – bisgambiglia_pa@univ-corse.fr
Marie-Laure NIVET – nivet_ml@univ-corse.fr
Evelyne VITTORI - vittori_e@univ-corse.fr

Università di Corsica Pasquale Paoli
Laboratoire Sciences pour l'Environnement UMR 6134 SPE
cnrs

Objectifs de ce cours

- Introduire l'atelier 3
  - Faire un point sur les structures itératives
    - Boucles **for**
    - Boucles **while**
    - Fonction **range()** pour la génération de séquences d'entiers
  - Introduire les types conteneurs séquences ordonnées qui sont des types structurés : list, tuple, str
  - Reparler de **mutabilité** (list) et **d'immutabilité** (tuple et str)

N'oubliez pas les futures "Battle" de vocabulaire...

En programmation comment faire pour répéter une/des opération(s) ?

Les structures itératives ou boucles
s'utilisent pour répéter plusieurs fois
l'exécution d'une partie du programme.

Contrôle de saisie

test = FAUX

Tant que !test :

entrée = saisie

if entrée ok :

test = TRUE

```
print("chiffre entre 1 et 5")
fin_test = False
while not (fin_test):
      monChiffre = int(input("chiffre ?"))
      if monChiffre >= 1 and monChiffre <=5:
            fin_test = True
else:
     print"bravo")
     #suite du programme
```

```
try :
    x = float(input("Entrer un nombre : "))
except ValueError :
    print("ce n'est pas un nombre ! valeur mise à un par défaut")
    x=1
```

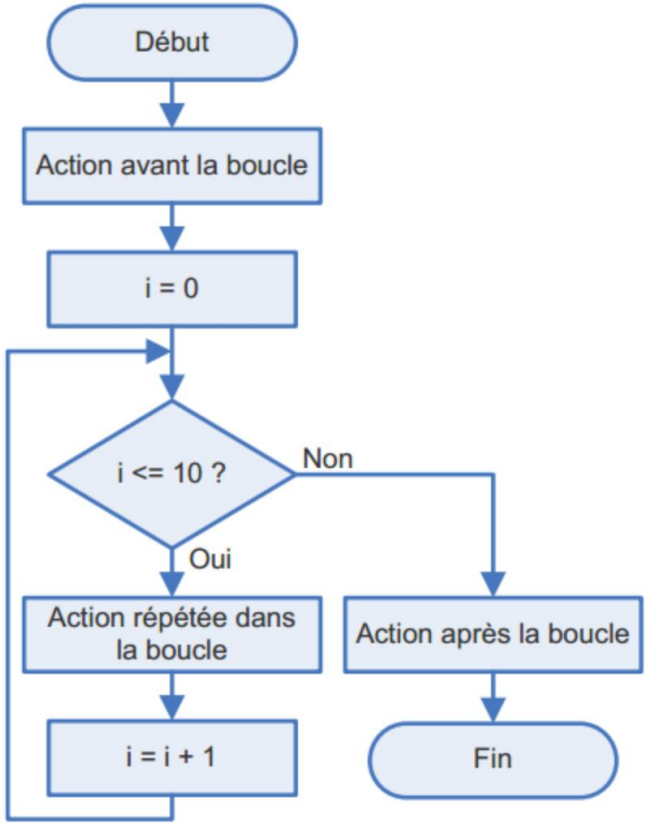
Boucles bornées et non bornées ?

- Boucle bornée : quand on sait combien de fois doit avoir lieu la répétition, on utilise généralement une boucle for dîtes pour.
- Boucle non bornée : si on ne connait pas à l'avance le nombre de répétitions, on choisit une boucle while dîtes tant que.

Déroulement

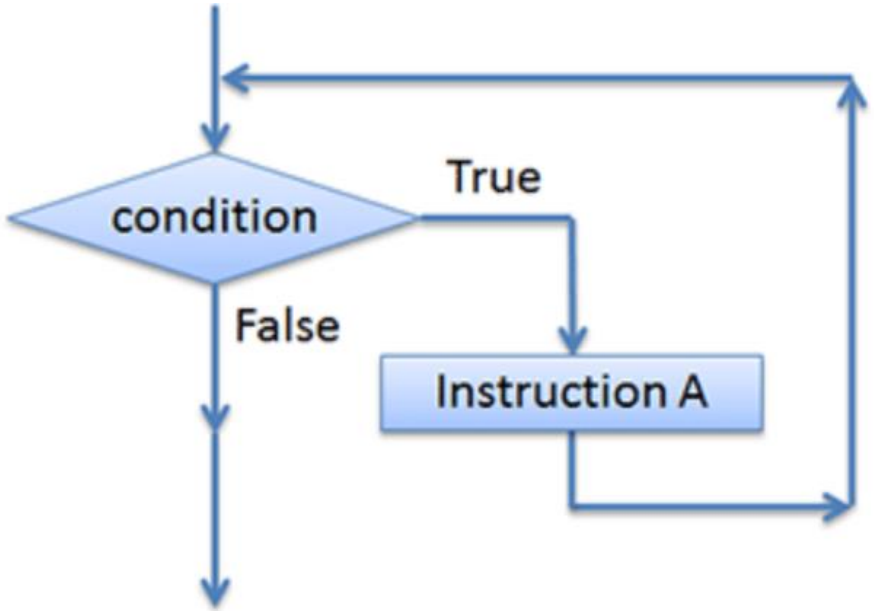
Pour

for



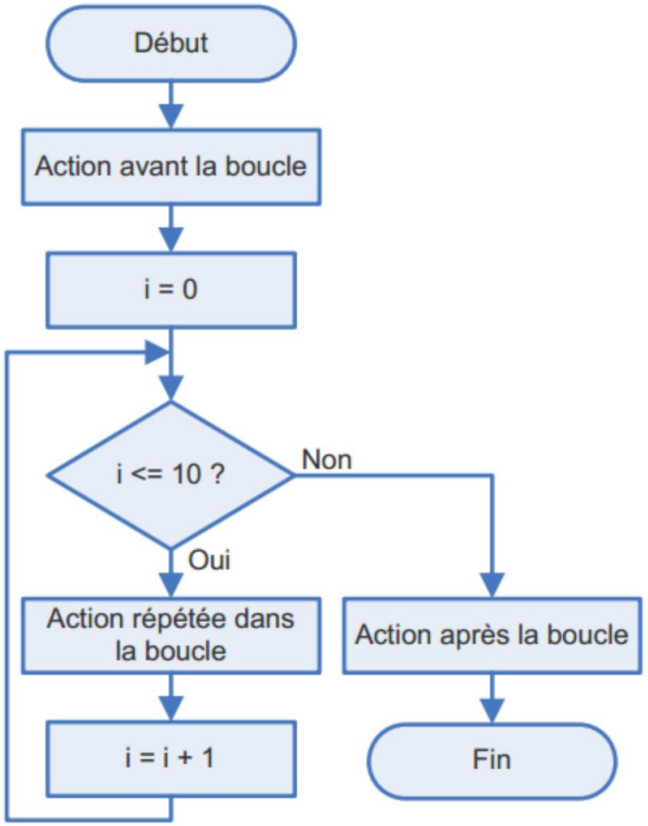
Tant que

while



Déroulement

Pour



For en python

```
for i in range(11): #range(10+1)
        #action dans la boucle
#action après la boucle
```

range(11) va renvoyer une **séquence** de nombre de 0 à 10
range(*start, stop, step*)

Pour bien comprendre testez les codes dans l'interpréteur

Séquences d'entiers

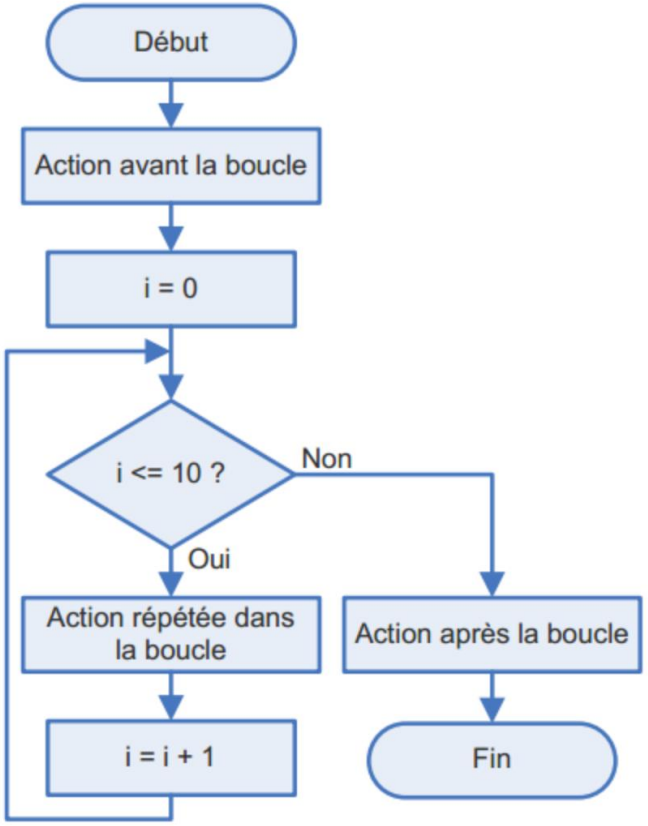
`range` ([*début*,] *fin* [,*pas*])

début défaut 0, *fin* non compris dans la séquence, *pas* signé et défaut 1

`range(5)` → 0 1 2 3 4 `range(2,12,3)` → 2 5 8 11
`range(3,8)` → 3 4 5 6 7 `range(20,5,-5)` → 20 15 10
`range(len(séq))` → *séquence des index des valeurs dans séq*
range fournit une séquence immutable d'entiers construits au besoin

Déroulement

Pour



Type liste, container séquence

For en python

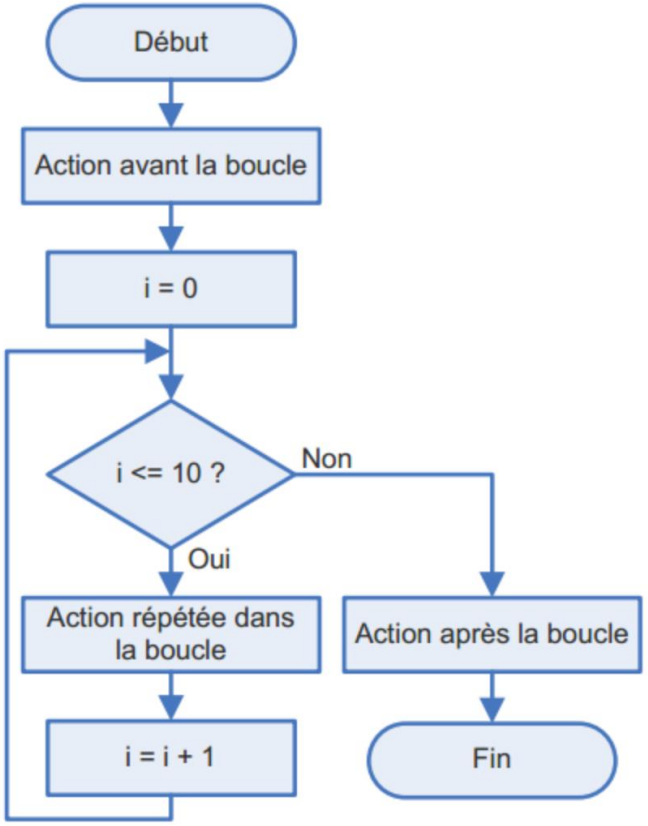
```
lst_test = [‘a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’j’, ‘k’]
borne_sup = len(lst_test) #11

for i in range(borne_sup):
        print lst_test[i]
```

étape
range
i
lst_test[i]

Déroulement

Pour



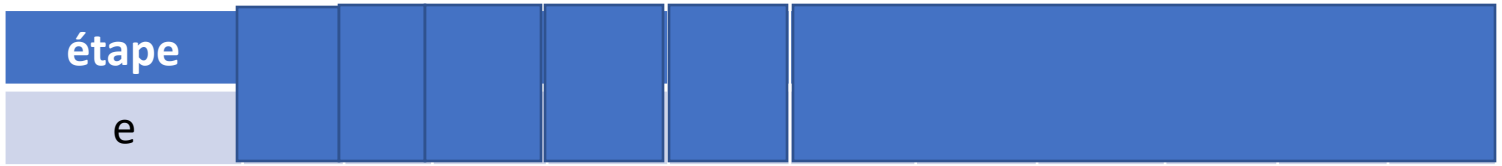
Type liste, containeur séquence

For en python

lst_test = [‘a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’j’, ‘k’]

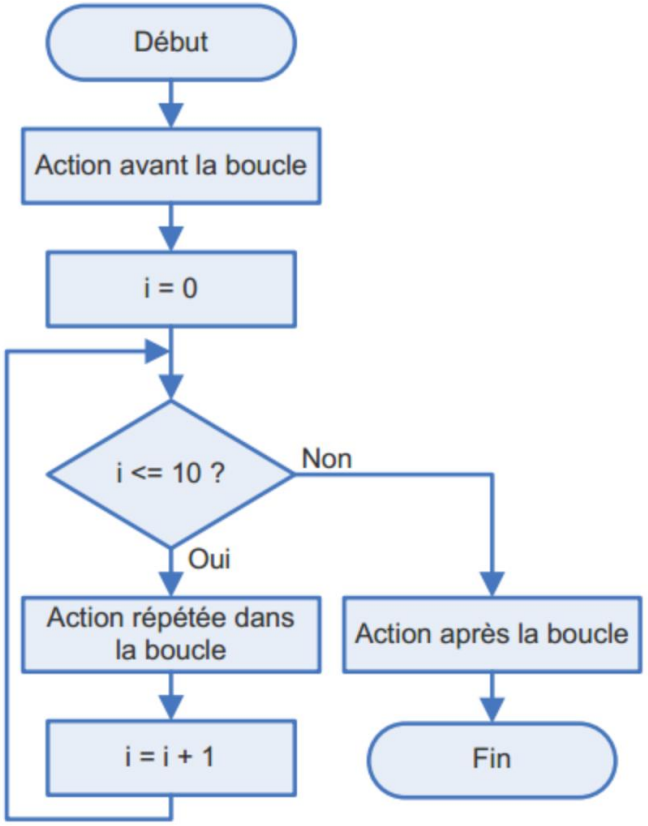
```
for e in lst_test:
        print (e)
```

Pour tous les éléments e de la séquence lst_test



Déroulement

Pour



Type liste, container séquence

For en python

```
lst_test = [‘a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’j’, ‘k’]

for i,e in enumerate(lst_test)
        print ("élément",e,"à l'indice",i)
```

étape						
i						
e						

Comment choisir ?

- En général, si on connaît avant de démarrer la boucle, le nombre d'itérations à exécuter, on choisit une boucle for. Au contraire, si la décision d'arrêter la boucle ne peut se faire que par un test, on choisit une boucle while.
- Il est toujours possible de remplacer une boucle for par une boucle while.

Exemple et transformation

Pour

```
for i in [0,1,2,3]
        print("i a pour valeur", i)
```

Tant que

```
i = 0
while i < 4:
        print("i a pour valeur", i)
        i = i + 1
```

Faire l’équivalent avec range()

Exemple en python

Pour

```
for i in [0, 1, 2, 3]:
        print("i a pour valeur", i)

for i in range(4):
        print("i a pour valeur", i)
```

Tant que

```
x = 1
while x < 10:
        print("x a pour valeur", x)
        x = x * 2
print("Fin")
```

Exemple en python

Pour

```
v = "Bonjour toi"
for lettre in v:
        print(lettre)
```

Tant que

Faire l’équivalent avec la boucle while

Petit jeu

```
def plusoumoins(min:int,max:int):
        randomNomber = random.randint(min,max)
        finduJeu = False
        coup = 0
        while not (finduJeu) and (coup < 11):
                monNbr = int(input("Entrez en nombre: "))
                if (monNbr > randomNomber):
                        print "trop grand"
                elif (monNbr < randomNomber):
                        print "trop petit"
                else:
                        print "Tu gagnes en ",coup,"coup(s)"
                finduJeu = True
                coup +=1
        if (coup > 10):
                print "perdu"
```

La clause else dans une boucle

Quelle sera la sortie de ce programme ? (simulez dans votre tête pas dans l'interpréteur...

- La clause else dans une boucle permet de définir un bloc d'instructions qui sera exécuté à la fin, seulement si la boucle s'est déroulée complétement sans être interrompue par un break.
- la clause else est exécutée lorsque la boucle se termine par épuisement de la liste (avec for) ou quand la condition devient fausse (avec while)

```
for n in range(2, 8):
    for x in range(2, n):
        if n % x == 0:
            print(n,"egale", x,"*",n/x)
            break
    else:
        print(n, "est un nombre premier")
```

Si cette boucle va à son terme cela veut dire que le nombre n n'a pas de diviseur

Rappel : un nombre est premier s'il est seulement divisible par 1 et lui-même

La clause else dans une boucle

```
for n in range(2, 8):
    for x in range(2, n):
        if n % x == 0:
            print(n,"égale", x,"*",n/x)
            break
    else:
        print(n, "est un nombre premier")
```

2 est un nombre premier

3 est un nombre premier

4 égale 2 * 2.0

5 est un nombre premier

6 égale 2 * 3.0

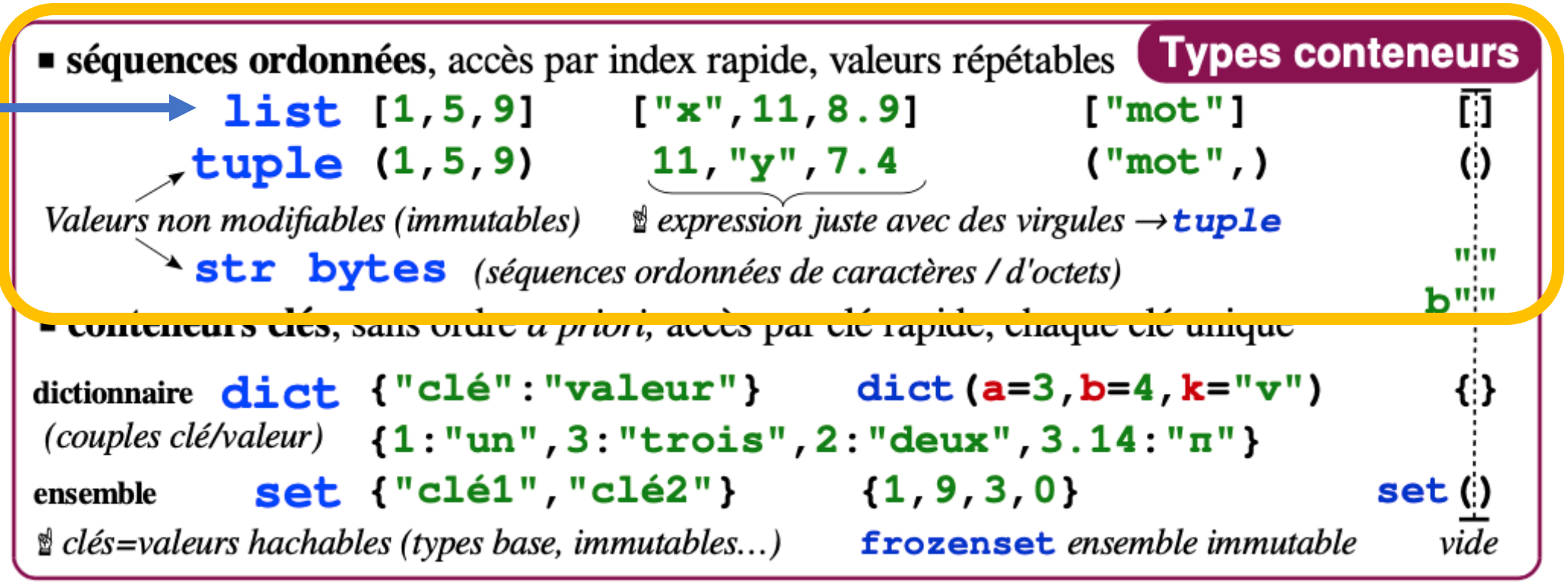
7 est un nombre premier

Les types conteneurs séquences ordonnées, types structurés

indexables, itérables

mutables

Les chaines sont des cas particuliers de listes non modifiables



Conteneurs indexables

pour les listes, tuples, chaînes de caractères, bytes…

Indexation conteneurs séquences

index négatif	−5	−4	−3	−2	−1
index positif	0	1	2	3	4
`lst=`	`[10,`	`20,`	`30,`	`40,`	`50]`
tranche positive	0	1	2	3	4	5
tranche négative	−5	−4	−3	−2	−1	

Nombre d'éléments

`len(lst)→5`

index à partir de 0
(de 0 à 4 ici)

Accès individuel aux **éléments** par `lst[`*index*`]`

`lst[0]→10` ⇒ *le premier* `lst[1]→20`

`lst[-1]→50` ⇒ *le dernier* `lst[-2]→40`

Sur les séquences modifiables (`list`), suppression avec `del lst[3]` *et modification par affectation* `lst[4]=25`

Accès à des **sous-séquences** par `lst[`*tranche début*`:`*tranche fin*`:`*pas*`]`

`lst[:-1]→[10,20,30,40]` `lst[::-1]→[50,40,30,20,10]` `lst[1:3]→[20,30]` `lst[:3]→[10,20,30]`

`lst[1:-1]→[20,30,40]` `lst[::-2]→[50,30,10]` `lst[-3:-1]→[30,40]` `lst[3:]→[40,50]`

`lst[::2]→[10,30,50]` `lst[:]→[10,20,30,40,50]` *copie superficielle de la séquence*

Indication de tranche manquante → à partir du début / jusqu'à la fin.

Sur les séquences modifiables (`list`), suppression avec `del lst[3:5]` *et modification par affectation* `lst[1:4]=[15,25]`

Les listes en python, objets mutables

Sont des tableaux dynamiques (leur taille évolue), des séquences d'objets hétérogènes.

l = [] #liste vide

l = [1, "a", bool]

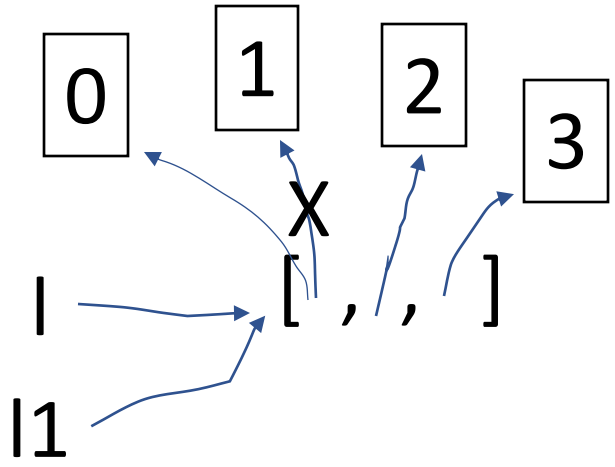
l = [1,2,3]

l1 = l #on a id(l1) == id(l)

l[0] = 0

#Attention l1 a également subi la modif, pour éviter cela

l2 = l[:] #Shallow copie



Opérations sur listes

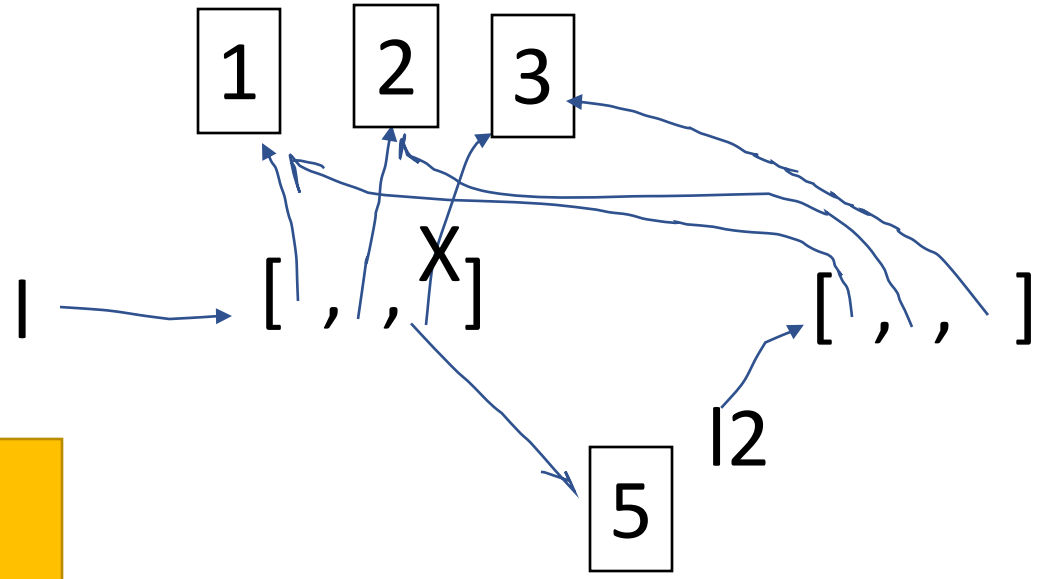
modification de la liste originale

`lst.append(val)`	ajout d'un élément à la fin
`lst.extend(seq)`	ajout d'une séquence d'éléments à la fin
`lst.insert(idx, val)`	insertion d'un élément à une position
`lst.remove(val)`	suppression du premier élément de valeur *val*
`lst.pop([idx])` →*valeur*	supp. & retourne l'item d'index *idx* (défaut le dernier)
`lst.sort()` `lst.reverse()`	tri / inversion de la liste sur place

Les listes en python, objets mutables

Sont des tableaux dynamiques (leur taille évolue), des séquences d'objets hétérogènes.

```
l = [1,2,3]
l2 = l[:] #Shallow copie
l[2] = 5
#l a été modifiée, pas l2
```



Attention, quand on travaille avec des listes de listes l=[1,[2]]… la copie de surface ne suffit plus !… Dessinez le pour comprendre

Cf.

```
import copy
copy.copy(c) → copie superficielle du conteneur
copy.deepcopy(c) → copie en profondeur du conteneur
```

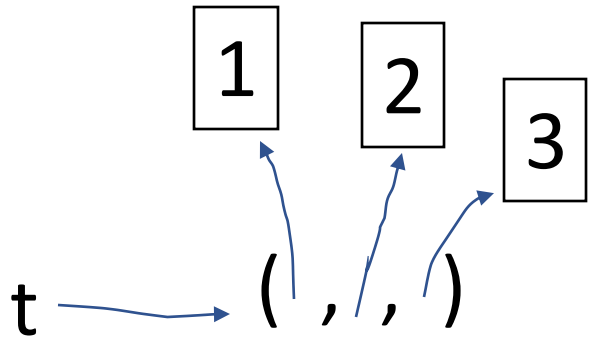
Tuples en python, sortes de listes hétérogènes et immutables, non modifiables

t = () #tuple vide, aucun intérêt car non mutable

t = (1,2,3)

t = (1,'un',True)

t = 'également', 'sans', 'parenthèse'



>>> print(t[0])

'également'

>>>t[0]='autre' TypeError: 'tuple' object does not support item assignment #Car non mutable

#Attention si le tuple contient des objets mutables…

Les chaînes en python, sortes de listes de caractères mais immutables

```
>>>s = ‘bonjour le monde’
>>>s.replace(‘le  monde’,’toto’)
‘bonjour toto’
>>>s
‘bonjour le monde’
```

Opérations sur chaînes

```
s.startswith(prefix[,début[,fin]])
s.endswith(suffix[,début[,fin]])   s.strip([caractères])
s.count(sub[,début[,fin]])   s.partition(sep) → (avant,sep,après)
s.index(sub[,début[,fin]])   s.find(sub[,début[,fin]])
s.is…()  tests sur les catégories de caractères (ex. s.isalpha())
s.upper()   s.lower()   s.title()   s.swapcase()
s.casefold()   s.capitalize()   s.center([larg,rempl])
s.ljust([larg,rempl])   s.rjust([larg,rempl])   s.zfill([larg])
s.encode(codage)   s.split([sep])   s.join(séq)
```

Opérations génériques sur conteneurs

```
len(c) → nb d'éléments
min(c)   max(c)   sum(c)
sorted(c) → list copie triée
val in c → booléen, opérateur in de test de présence (not in d'absence)
enumerate(c) → itérateur sur (index, valeur)
zip(c1,c2…) → itérateur sur tuples contenant les éléments de même index des ci
all(c) → True si tout élément de c évalué vrai, sinon False
any(c) → True si au moins un élément de c évalué vrai, sinon False
c.clear() supprime le contenu des dictionnaires, ensembles, listes
```

*Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les **clés**.*

*Spécifique aux **conteneurs de séquences ordonnées** (listes, tuples, chaînes, bytes…)*

```
reversed(c) → itérateur inversé   c*5→ duplication   c+c2→ concaténation
c.index(val) → position   c.count(val) → nb d'occurences
```

```
import copy
copy.copy(c) → copie superficielle du conteneur
copy.deepcopy(c) → copie en profondeur du conteneur
```

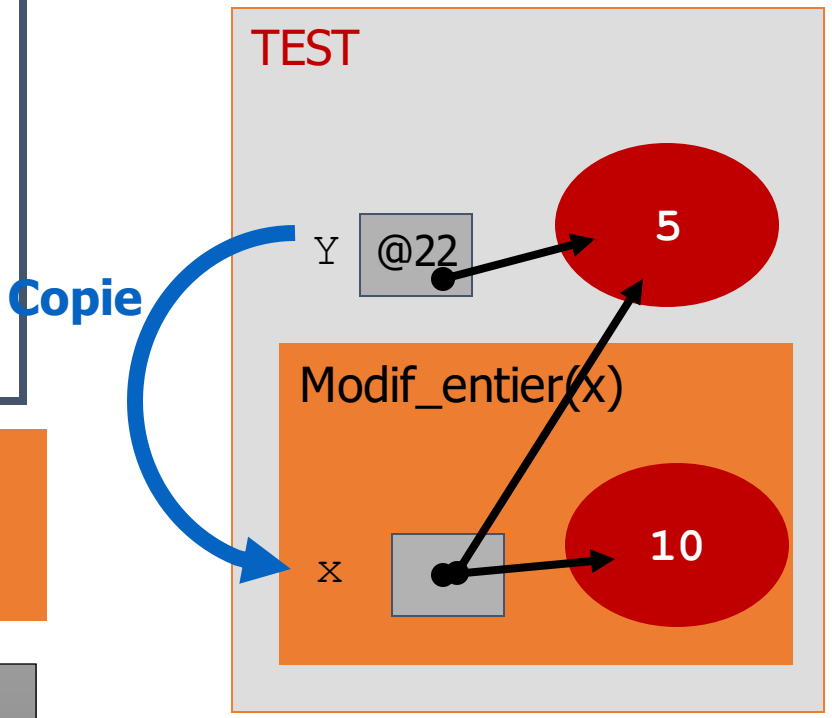
Fonctions en python

Passage d'un paramètre de type entier

Absence d'effet de bord

```
def modif_entier(X) :
    X=10

#TEST
Y=5
print("Avant appel Y= " +str(Y))
modif_entier(Y)
print("Aprés appel Y= " +str(Y))
}
```



Aucun effet de bord possible pour les objets immutables

Avant appel Y= 5
Apres appel Y= 5

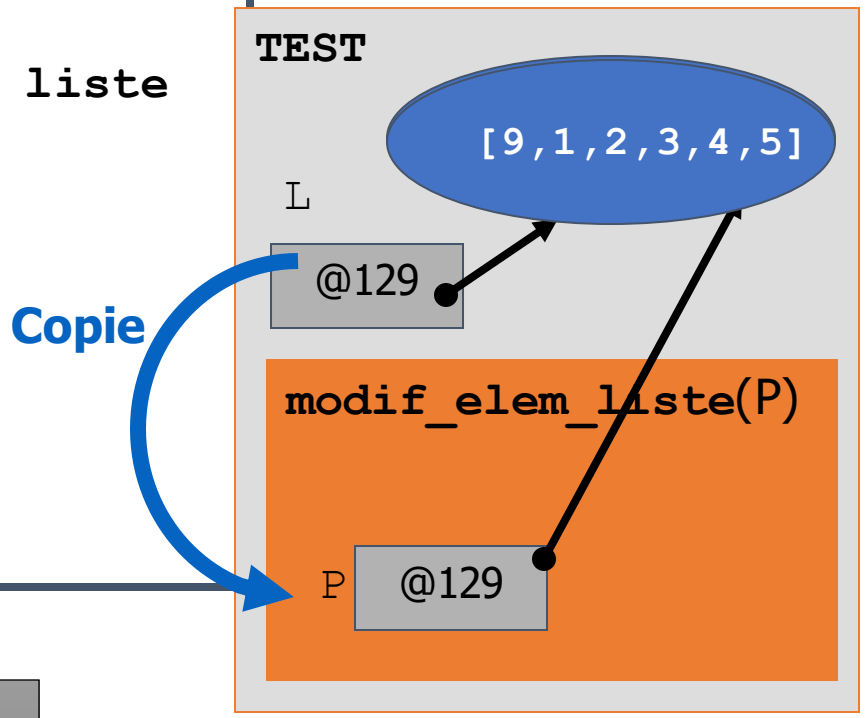
Fonctions en python

Passage d'un paramètre de type liste

Effet de bord

```
def modif_elem_liste(P) :
    #modifie le 1er élément d'une liste
    if len(P)!=0 :
        P[0] = 9

L=[0,1,2,3,4,5]
print("Avant appel L= " +str(L))
modif_elem_liste(L)
print("Apres appel L= " +str(L))
```



Avant appel L= [0, 1, 2, 3, 4, 5]
Apres appel L= [9, 1, 2, 3, 4, 5]

Passage par valeur, mutabilité et immutabilité

```
def func(x, y, z):
    x    = 27
    y[0] = 'foofoo'
    print('En local, dans func, la liste y ',y)
    y = [4,5]
    print('En local après ré-affectation, la liste y ',y)
    # z est non mutable, on essaie pas d'en modifier les composants
    print('En local, dans func, le tuple z ',z)
    z = (8, 10, 12)
    print('En local après ré-affectation, dans func, le tuple z ',z)
```

Que se passe-t-il ici ?

```
x = 1                #non mutable
y = [2, 3, 5, 7]     #mutable
z = (1, 2, 3)        #non mutable

func(x, y, z)

print(x, y, z)
```

Passage par valeur, mutabilité et immutabilité

```
def func(x, y, z):
    x    = 27
    y[0] = 'foofoo'
    print('En local, dans func, la liste y ',y)
    y = [4,5]
    print('En local après ré-affectation, la liste y ',y)
    # z est non mutable, on essaie pas d'en modifier les composants
    print('En local, dans func, le tuple z ',z)
    z = (8, 10, 12)
    print('En local après ré-affectation, dans func, le tuple z ',z)
```

Que se passe-t-il ici ?

```
x = 1                #non mutable
y = [2, 3, 5, 7]     #mutable
z = (1, 2, 3)        #non mutable

func(x, y, z)

print(x, y, z)
```

```
en local dans funct y  ['foofoo', 3, 5, 7]
en local dans funct après réacfectation y  [4, 5]
En local dans fonct tuple z (1, 2, 3)
en local aprs reafect z (8, 10, 12)
1 ['foofoo', 3, 5, 7] (1, 2, 3)
```

Liens

- Images diapo 4 :
  - https://www.electro-info.ovh/les-structures-algorithmiques-de-base#ph30
  - https://courspython.com/boucles.html

- Exemple de code :
  - https://docs.python.org/3/tutorial/controlflow.html
  - https://courspython.com/boucles.html

- En savoir plus sur les exceptions :
  - https://docs.python.org/3/tutorial/errors.html#handling-exceptions