Introduction à Python

V. Monbet
(document fortement inspiré du livre de R. Cordeau
et des supports de cours de J.-P. Becirspahic)

UFR de Mathématiques
Université de Rennes 1

15 juillet 2018

Outline

Définition d'une fonction

On définit une fonction à l'aide du mot clé def :

```
def nomdelafcn (liste de parametres):
    bloc ..........................
    d instructions ................
    a realiser ....................
```

Traditionnellement on distingue deux types de routines : les procédures ne retournent pas de résultat et se contentent d'agir sur l'environnement, les fonctions retournent un résultat. En PYTHON la distinction n'existe pas réellement : les procédures sont des fonctions qui retournent la valeur None.
Un résultat retourné par une fonction peut être réutilisé dans un calcul, à 'inverse d'une procédure :

```
In [3]: 1 + len("45")
Out[3]: 3

In [4]: 1 + print(45)
TypeError: unsupported operand type(s)
    for +: 'int' and 'NoneType'
```

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule. Pour définir la fonction $(x, y) \mapsto \sqrt{x^2 + y^2}$:

```
from numpy import sqrt

def norme(x, y):
    return sqrt(x**2 + y**2)
```

Une fois définie, une fonction s'utilise à l'instar de toute autre fonction prédéfinie :

```
In [1]: norme(3, 4)
Out[1]: 5.0
```

En effet, $\sqrt{3^2 + 4^2} = 5$.

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule. Pour définir la fonction $(x, y) \mapsto (x^k + y^k)^{1/k}$:

```
from numpy import sqrt

def norme(x, y,k):
    return(x**k + y**k)**(1/k)
```

Avec cette nouvelle définition, on a :

```
In [2]: norme(3, 4, 2)
Out[2]: 5.0
In [3]: norme(3, 4, 3)
Out[3]: 4.497941445275415
```

En effet, $(3^3 + 4^3)^{1/3} = 4.4979$.

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule. Pour définir la fonction $(x, y) \mapsto (x^k + y^k)^{1/k}$:

```
from numpy import sqrt

def norme(x, y,k):
    return(x**k + y**k)**(1/k)
```

Avec cette nouvelle définition, on a :

```
In [2]: norme(3, 4, 2)
Out[2]: 5.0

In [3]: norme(3, 4, 3)
Out[3]: 4.497941445275415

In [4]: norme(3, 4)
TypeError: norme() takes
    exactly 3 arguments (2 given)
```

Une erreur est déclenchée dès lors que le nombre d'arguments donnés est incorrect.

Arguments d'une fonction
Arguments optionnels

Il est possible de préciser les valeurs par défaut que doivent prendre certains arguments.

```
from numpy import sqrt

def norme(x, y,k=2):
    return(x**k + y**k)**(1/k)
```

Si on omet de préciser le troisième paramètre, ce dernier sera égal à 2 :

```
In [5]: norme(3, 4, 3)
Out[5]: 4.497941445275415

In [6]: norme(3, 4)
Out[6]: 5.0
```

Il est préférable de nommer les arguments optionnels pour éviter toute ambiguïté :

```
In [7]: norme(3, 4, k=3)
Out[7]: 4.497941445275415
```

Arguments d'une fonction
Arguments optionnels

C'est le cas de la fonction **print** qui possède deux paramètres optionnels sep (valeur par défaut : `' '`) qui est inséré entre chacun des arguments de la fonction et end (valeur par défaut : `'\n'`) qui est ajouté à la fin du dernier des arguments :

```
In [8]: print(1, 2, 3, sep='+', end='=6\n')
1+2+3=6
```

Portée des variables

Les variables définies dans une fonction ne sont accessibles que dans la fonction elle-même. De telles variables sont qualifiées de locales, par opposition aux variables globales. Si on souhaite modifier le contenu d'une variable globale à l'intérieur du bloc d'instructions d'une fonction, il faut utiliser l'instruction **global** pour déclarer celles des variables qui doivent être traitées globalement.

```
def h():
    global a # declaration
        #d une variable globale
    a = 2
    return a

In [17]: h()
Out[17]: 2
In [18]: a # la variable definie ligne 9
    #a bien ete modifiee
Out[18]: 2
```

En règle générale, il est conseillé de peu faire usage de variables globales.

Portée des variables - Exercice

```
def f():
    global a
    a = a + 1
    return a
```

```
def g():
    a = 1
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

Que produit le script suivant ?

```
a = 1
print(f(), a)
print(a, f())
print(a, g())
print(a, h())
```

Portée des variables - Exercice

```
def f():
    global a
    a = a + 1
    return a
```

```
def g():
    a = 1
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

Que produit le script suivant ?

```
a = 1
print(f(), a)
print(a, f())
print(a, g())
print(a, h())
```

Résultat

```
2 2
2 3
3 2
UnboundLocalError: local variable 'a' referenced
      before assignment
```

Portée des variables - Exercice

```
def f():
    global a
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

On utilise la fonction `dis` du module du même nom qui désassemble le bytecode :

```
dis.dis(f)
  3   0 LOAD_GLOBAL    0 (a)
      3 LOAD_CONST     1 (1)
      6 BINARY_ADD
      7 STORE_GLOBAL   0 (a)

  4  10 LOAD_GLOBAL    0 (a)
     13 RETURN_VALUE
```

```
dis.dis(h)
 14   0 LOAD_FAST      0 (a)
      3 LOAD_CONST     1 (1)
      6 BINARY_ADD
      7 STORE_FAST     0 (a)

 15  10 LOAD_FAST      0 (a)
     13 RETURN_VALUE
```

Dans la fonction h, la variable locale a est référencée (LOAD_FAST) avant d'être assignée (STORE_FAST). Il n'y a pas d'erreur dans f puisque la variable globale a est déjà assignée lorsqu'elle est référencée par LOAD_GLOBAL.

Outline

Séquences

Définition

Une séquence est un conteneur ordonné d'éléments indicés par des entiers.

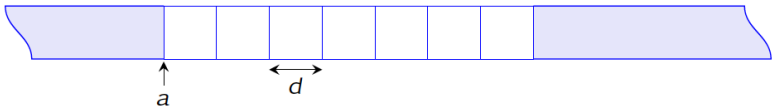
PYTHON dispose de trois types prédéfinis de séquences : - les chaînes de caractères ; - les listes ; - les tuples.

Outline

Les listes

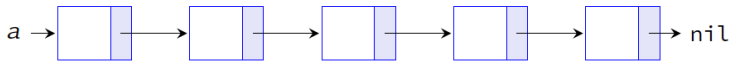
Les listes sont les principales structures de données en PYTHON. C'est une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les tableaux et les listes chaînées.

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire :



Les tableaux existent en Python : c'est la classe `array` fournie par la bibliothèque NUMPY.

Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :



Les listes n'existent pas en PYTHON : la classe **`list`** n'est pas une liste chaînée mais une structure de données qui concilie les avantages des tableaux et des listes chaînées.
c'est une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant.

Construction d'une liste

Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

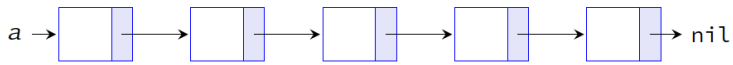
```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
print('abc' in a) # True
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

On accède à chaque élément de la liste par son index, qui débute à 0.

```
In [1]: a[2]
Out[1]: 'abc'

In [2]: b[3][1]
Out[2]: 2
```



Construction d'une liste

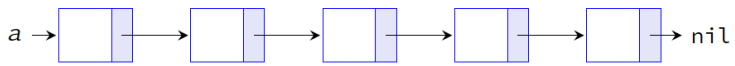
Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

Lorsque l'index est négatif, le décompte est pris en partant de la fin. Ainsi, les éléments d'indices -k et ℓ-k sont les mêmes.

```
In [4]: a[-2]
Out[4]: 'de'
```



Slicing

PYTHON permet le « découpage en tranches » : si l est une liste, alors l[i :j] est une nouvelle liste constituée des éléments dont les index sont compris entre i et j - 1 :

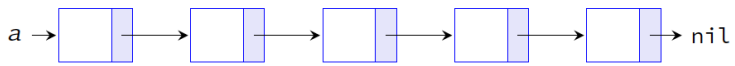
```
In [5]: a[1:5]
Out[5]: [1, 'abc', 4.5, 'de']

In [6]: a[1:-1]
Out[6]: [1, 'abc', 4.5, 'de']
```

Lorsque i est absent, il est pris par défaut égal à 0 ; lorsque j est absent, il est pris par défaut égal à la longueur de la liste.

```
In [7]: a[:4]
Out[7]: [0, 1, 'abc', 4.5]

In [8]: a[-3:]
Out[8]: [4.5, 'de', 6]
```



À retenir l[:n] calcule la liste des n premiers éléments et l[-n :] la liste des n derniers.

Slicing

Sélection partielle

La syntaxe `lst[debut:fin]` possède un troisième paramètre (égal par défaut à 1) indiquant le pas de la sélection :

```
In [9]: l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
In [10]: l[2:9:3]
Out[10]: [2, 5, 8]
In [11]: l[3::2]
Out[11]: [3, 5, 7, 9]
In [12]: l[:-1:2]
Out[12]: [0, 2, 4, 6, 8]
```

Il est possible de choisir un pas négatif :

```
In [13]: l[-1:0:-1]
Out[13]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Création d'une liste

par énumération

Les listes sont de type **`list`** et la fonction **`list`** convertit lorsque c'est possible un objet d'un certain type vers le type **`list`**. C'est le cas en particulier des énumérations produites par la fonction **`range`** :

```
In [14]: list(range(11))
Out[14]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [15]: list(range(13, 2, -3))
Out[15]: [13, 10, 7, 4]
```

Création d'une liste

par compréhension

Il est possible de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique.

$\{x \in [0, 10] | x^2 \leq 50\}$

```
In [17]: [x for x in range(11) if x * x <= 50]
Out[17]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Produit cartésien de deux listes :

```
In [18]: a, b = [1, 3, 5], [2, 4, 6]

In [19]: [(x, y) for x in a for y in b]
Out[19]: [(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6),
(5, 2), (5, 4),(5, 6)]
```

Opérations sur les listes

L'opération de concaténation se note + :

```
In [1]: [2, 4, 6, 8] + [1, 3, 5, 7]
Out[1]: [2, 4, 6, 8, 1, 3, 5, 7]
```

L'opérateur de duplication se note * ; si lst est une liste et n un entier alors lst * n est équivalent à lst + lst + ... + lst (n fois).

```
In [2]: [1, 2, 3] * 3
Out[2]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Suppression d'un ou plusieurs éléments :

```
In [9]: l = list(range(11))
In [10]: del l[3:6]
In [11]: l
Out[11]: [0, 1, 2, 6, 7, 8, 9, 10]
```

Mutation d'une liste

Une liste est un objet mutable, c'est à dire modifiable.
Suppression d'un ou plusieurs éléments :

```
In [9]: l = list(range(11))
In [10]: del l[3:6]
In [11]: l
Out[11]: [0, 1, 2, 6, 7, 8, 9, 10]
```

Insertion d'un élément :

```
In [12]: l = ['a', 'b', 'c', 'd']
In [13]: l.append('e')
In [14]: l
Out[14]: ['a', 'b', 'c', 'd', 'e']
In [15]: l.insert(2, 'x')
In [16]: l
Out[16]: ['a', 'b', 'x', 'c', 'd', 'e']
```

Quelques méthodes associées aux listes

```
In [17]: l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
In [18]: l.remove(4) # retire les 4
In [19]: l
Out[19]: [1, 2, 3, 5, 1, 2, 3, 4, 5]
In [20]: l.pop(-1) # supprime l element d indice i et le retourne
Out[20]: 5
In [21]: l
Out[21]: [1, 2, 3, 5, 1, 2, 3, 4]
In [22]: l = [1, 2, 3, 4, 1, 2, 3, 4]
In [23]: l.reverse()
In [24]: l
Out[24]: [4, 3, 2, 1, 4, 3, 2, 1]
In [25]: l.sort()
In [26]: l
Out[26]: [1, 1, 2, 2, 3, 3, 4, 4]
```

Parcours d'une liste

Parcours complet par boucle énumérée

La syntaxe **for** x **in** seq : permet de parcourir tous les éléments x d'une séquence seq.
Exemple : calcul de la somme des éléments d'une liste (ou d'un tuple).

```
def somme(l):
    s = 0
    for x in l:
        s += x
    return s
```

Calcul de la moyenne $\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$

```
def moyenne(l):
    s = 0
    for x in l:
        s += x
    return s/len(l)
```

Parcours d'une liste

Exercices

1. Écrire une fonction pour calculer la variance des éléments d'une liste.
2. Écrire une fonction pour trouver le maximum des éléments d'une liste.
3. Écrire une fonction pour trouver l'indice du maximum des éléments d'une liste.

Parcours d'une liste

Calcul du maximum

Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

Calcul de l'indice de l'élément maximal : on parcourt la liste des indices.

```
def indice_max(l):
    i, m = 0, l[0]
    for k in range (1, len(l)):
        if l[k] > m:
            i, m = k, l[k]
    return i
```

Parcours d'une liste

Calcul du maximum

Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

La fonction **enumerate** renvoie une liste de tuples (indice, valeur) lorsqu'on l'applique à un objet énumérable.

```
def indice_du_max(l):
    i, m = 0, l[0]
    for (k, x) in enumerate(l):
        if x > m:
            i, m = k, x
    return i
```

Parcours d'une liste

Parcours incomplet (recherche d'un élément)

Un problème fréquent : déterminer la présence ou non d'un élément x dans une liste l.

Première méthode : parcourir la liste par une boucle conditionnelle.

```
def cherche(x, l):
    k, rep = 0, False
    while k < len(l) and not rep:
        rep = l[k] == x
        k += 1
    return rep
```

Deuxième méthode : interrompre une boucle énumérée dès l'élément trouvé.

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

Exercices

(1) Écrire une fonction qui renvoie le plus petit élément d'une liste.

(2) Écrire une fonction qui renvoie la liste triée par ordre croissant du dernier élément des tuples.
`Liste = [(2, 5), (1, 2), (4, 4), (2, 3), (2, 1)]`
Résultat : `[(2, 1), (1, 2), (2, 3), (4, 4), (2, 5)]`

```
?sorted
Signature: sorted(iterable, /, *, key=None, reverse=False)
Docstring: Return a new list containing all items
from the iterable  in ascending order.
A custom key function can be supplied to customize the sort order,
and the reverse flag can be set to request the result
in descending order.
```

(3) Écrire une fonction qui renvoie le nombre de chaines telles que la chaine est de longueur supérieure à 2 et ses premier et dernier caractères sont égaux.
`Liste = ['abc', 'xyz', 'aba', '1221']`
Résultat : 2

(4) Écrire un programme qui supprime les duplicats dans une liste.

Banque d'exercices : https ://www.w3resource.com/python-exercises/list/

Outline

Tuples

Définition

Un **tuple** est une collection ordonnée et non mutable d'éléments éventuellement hétérogènes.

```
mon_tuple = ("a", 2, [1, 3])
```

- Les tuples s'utilisent comme les listes mais leur parcours est plus rapide ;
- Ils sont utiles pour définir des constantes.

Tuples

Méthodes

```
tup = tuple("trouver un index")
print(tup)
print(tup.index("v"))
print(tup.index("v",2))
print(tup.index("e",7,15))
```

```
('t', 'r', 'o', 'u', 'v', 'e', 'r', ' ', 'u', 'n', ' ', 'i', 'n', 'd',
4
4
14
```

Séquences et mutabilité

Séquence mutable : on peut modifier ou supprimer les éléments de cette séquence (listes).

```
In [1]: lst = [1, 2, 3]
In [2]: lst[0] = 4
In [3]: lst
Out[3]: [4, 2, 3]
```

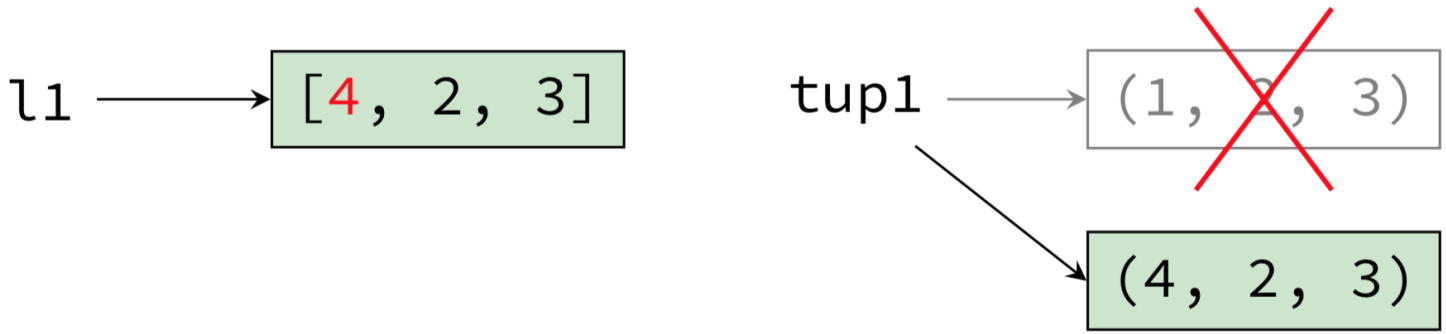
Séquence non mutable : on ne peut modifier ou supprimer les éléments de cette séquence (chaînes de caratères et tuples).

```
In [4]: tup = (1, 2, 3)
In [5]: tup[0] = 4
TypeError: 'tuple' object does not support item assignment
In [6]: chr = 'abc'
In [7]: chr[0] = 'd'
TypeError: 'str' object does not support item assignment
```

Séquences

Rappel le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
 In [7]: l1[0] = 4
 In [8]: id(l1)
Out[8]: 4448440520
In [9]: tup1 = (4, 2, 3)
In [10]: id(tup1)
Out[10]: 4448266640
```

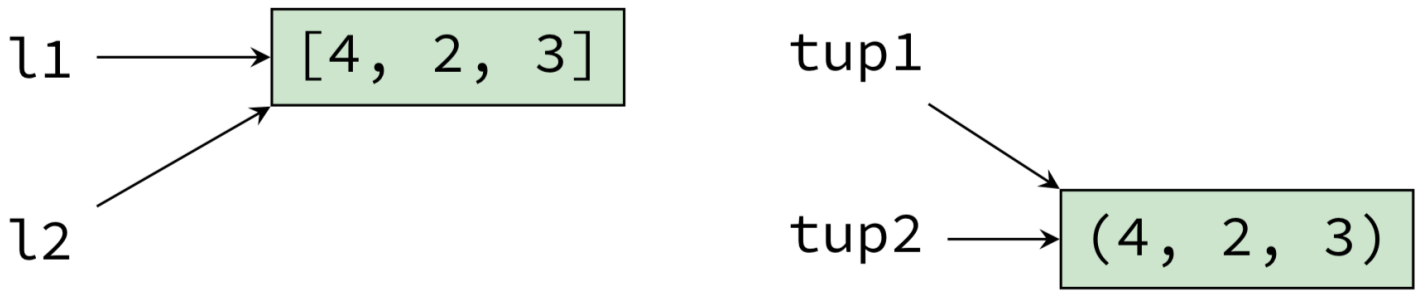


Une liste est mutable, on peut la modifier sans pour autant créer une nouvelle référence vers l'objet, contrairement à un tuple.

Séquences

Rappel : le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
In [11]: l2 = l1
In [12]: id(l1) == id(l2)
Out[12]: True
In [13]: tup2 = tup1
In [14]: id(tup1) == id(tup2)
Out[14]: True
```

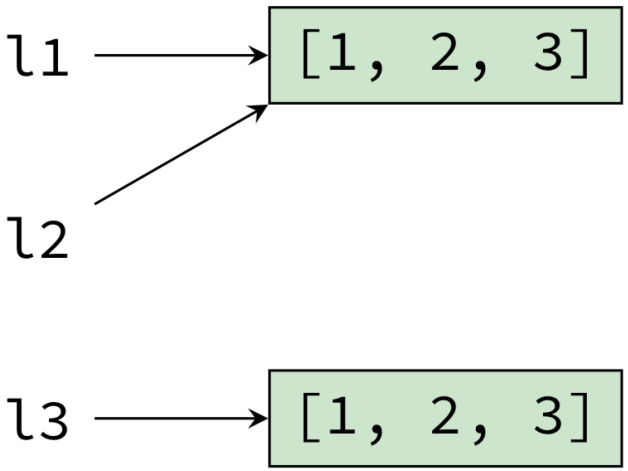


lorsque `var1` est une variable, l'instruction `var2 = var1` crée un nouveau référencement vers le même emplacement en mémoire.

Séquences

Rappel : le caractère = crée un lien symbolique (un pointeur) entre le nom choisi et l'emplacement en mémoire contenant la valeur.

```
In [15]: l1[0] = 1
In [16]: l2
Out[16]: [1, 2, 3]
In [17]: tup1 = (1, 2, 3)
In [18]: tup2
Out[18]: (4, 2, 3)
```

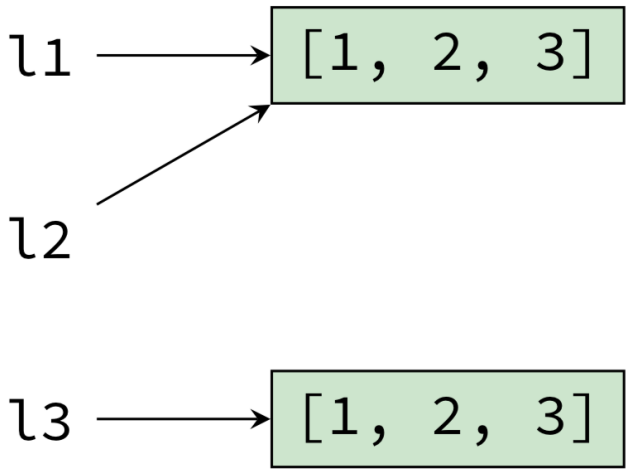


L"nstruction `l1[0] = 1` modifie aussi la variable `l2` puisque ces deux noms référencent la même adresse.

Copie d'un objet mutable

Un nouveau référencement n'est pas suffisant pour copier un objet mutable ; il est nécessaire de le recréer entièrement, par exemple à l'aide du slicing `[:]`.

```
In [19]: l3 = l1[:]
In [20]: id(l1) == id(l3)
  Out[20]: False
```



De manière équivalente on peut écrire :

`l3=l1.copy()` ou `l3=[x for x in l1]`

Tuple, exercices

1. Écrire un programme pour remplacer la dernière valeur de tous les tuples d'une liste.
   Par exemple,
   - liste d'entrée : `[(10, 20, 40), (40, 50, 60), (70, 80, 90)]` - liste attentdue en sortie : `[(10, 20, 100), (40, 50, 100), (70, 80, 100)]`

2. Écrire un programme qui compte les éléments d'une liste jusqu'à rencontrer un tuple.
   Indication on utilisera la fonction **`isinstance`**`(x,`**`tuple`**`)` qui retourne `True` si `x` est un tuple.
   Entrée : `lst = [10,20,30,(10,20),40]`

Outline

Les dictionnaires

Définition

Un **dictionnaire** **`dict`** est un type de données permettant de stocker des couples `cle : valeur`, avec un accès très rapide à la valeur à partir de la clé, la clé ne pouvant être présente qu'une seule fois dans le tableau.
Il possède les caractéristiques suivantes :
- l'opérateur d'appartenance d'une clé (**`in`**) ;
- la fonction taille (**`len`**`()`) donnant le nombre de couples stockés ;
- il est itérable (on peut le parcourir) mais n'est pas ordonné.

Les dictionnaires

```
# insertion de cles/valeurs une a une
d1 = {} # dictionnaire vide
d1["nom"] = 3
d1["taille"] = 176
print(d1) # {"nom": 3, "taille": 176} # definition en extension
d2 = {"nom": 3, "taille": 176}
print(d2) # {"nom": 3, "taille": 176} # definition en intension
d3 = {x: x**2 for x in (2, 4, 6)}
print(d3) # {2: 4, 4: 16, 6: 36}
# utilisation de parametres nommes
d4 = dict(nom=3, taille=176)
print(d4) # {"taille": 176, "nom": 3}
# utilisation d'une liste de couples cles/valeurs
d5 = dict([("nom", 3), ("taille", 176)])
print(d5) # {"nom": 3, "taille": 176}
```

Quelques méthodes pour les dictionnaires

```
tel = {"jack": 4098, "sape": 4139}
tel["guido"] = 4127
print(tel) # {"sape": 4139, "jack": 4098, "guido": 4127}
print(tel["jack"]) # 4098
del tel["sape"]
tel["irv"] = 4127
print(tel) # {"jack": 4098, "irv": 4127, "guido": 4127}
print(list(tel.keys())) # ["jack", "irv", "guido"]
print(sorted(tel.keys())) # ["guido", "irv", "jack"]
print(sorted(tel.values())) # [4098, 4127, 4127]
print("guido" in tel, "jack" not in tel) # True False
tel.update= {"jack": 4098, "sape": 4139, "tony": 4532}
```

Dictionnaires, exercices

Écrire un programme pour définir un dictionnaire (par exemple `d = 1: 2, 3: 4, 4: 3, 2: 1, 0: 0` puis

1. le trier par ordre ascendant (puis descendant) par ses valeurs ;
2. lui ajouter une clé avec ses valeurs (par exemple `0 (3,0)20: (3,0)` ;
3. vérifier que la clé "3" appartient au dictionnaire.