Chapitre 1

´El´ements de base de programmation

I. Variables élémentaires

Le langage python possède de nombreux types de variables, nous nous restreindrons pour le moment à la manipulation de certaines d'entre elles. On distinguera les variables définissant un objet seul.

1 - Les variables numériques

comme

— les variables numériques entières (type int) :

```
1  n = 2
```

dans un souci de relecture, les variables entières sont stockées sous les lettres : i,j,k,l,m,p,n.

— les variables numériques flottantes (type float) :

```
1  x = 2
```

dans un souci de relecture, les variables entières sont stockées sous les lettres : x,z,u,v.

Remarque 1 :
On bannira les noms de variables : a,b,c....

2 - Les variables booléennes

Les variables booléennes sont définies comme telles ou résultat d'une expression :

```
1  bool1 = False
2  bool2 = n==x
```

Ces variables seront utilisées comme critère d'exécution pour des structures conditionnelles (if, elif while...)

Les opérations élémentaires sur les nombres sont contenues dans Python :

— + : addition
— - : soustraction
— * : multiplication
— / : division
— ** : puissance

Les syntaxes suivantes permettent des calculs mois fréquents mais tout aussi importants :

— // : division entière
— % : reste de division euclidienne
— += : ajout (a = a + 3)
— *=, /=, -=...

3 - Les collections d'éléments

Des variables définissant une collection d'éléments comme :

— les chaînes de caractères définies entre guillements :

```
chaine = "bonjour MPSI"
```

— les listes définies entre crochets :

```
liste = [1,2,3,4]
```

— les tuples définies entre parenthèses :

```
t = (1,2,3,4)
```

— les dictionnaires définis entre accolades :

```
dico = {"eleve": 48, "prof" : 1}
```

Nous verrons au fur et à mesure les spécificités de chacune de ces collections.

II. L'objet List

1 - Caractéristiques

Définition : *Liste*

Sous Python, on peut définir une liste comme une collection d'éléments ordonnés, séparés par des virgules, l'ensemble étant enfermé dans des crochets.

Propriété :

Les éléments des listes peuvent être de types différents : texte, nombre, liste...

```
liste_1 = ['lundi',  1800,[1,2,3]]
```

Propriété :

Les éléments d'une collection sont accessibles à partir de leur indice **commençant par 0** :

```
liste1 [0]
>>> 'lundi'
liste1 [1]
>>> 1800
liste1 [2]
>>> [1,2,3]
```

Remarque 2 :

Les tuples et les chaînes de caractères obéissent aux mêmes caractéristiques, les éléments sont indicés de la même façon.

Propriété :

Une liste possède une taille définie par la fonctin `len`

```
>>>liste = ['lundi',  1800,[1,2,3]]
>>>len(liste)
    3
```

2 - Manipulation

Propriété :

Une liste est *mutable*, c'est à dire que ces éléments peuvent être modifiés.

```
liste_1[0] =  'mardi'
['mardi',  1800,[1,2,3]]
```

On peut ajouter des éléments à la fin d'une liste avec la méthode `.append`

```
liste_1.append('jeudi')
>>> ['mardi',  1800,[1,2,3],'jeudi']
```

Exemple 1 :

On considère une liste composés d'une suite de deux éléments $L = [[x_0,y0],..[x_{n-1},y_{n-1}]]$. Proposer un code permettant d'obtenir deux listes `X` et `Y` comprenant la suite des abscisses et des ordonnées : $X = [x_0,..,x_{n-1}]$ et $Y = [y_0,..,y_{n-1}]$.

```
X, Y = [], []
for coord in L:
    x,y = coord
    X.append(x)
    Y.append(y)
```

On peut enlever l'élément avec la méthode `.pop`

```
liste_1.pop()
>>> ['mardi',  1800,[1,2,3]]
```

3 - Parcours

a) Par itérateur

Propriété :

Les collections citées sont des objets **itérables**, c'est à dire un objet dont on peut parcourir les valeurs. .

On peut parcourir les éléments d'un itérable avec la construction :

```
for element in iterable:
    # bloc dans lequel
    # on dispose d element
```

Comme pour les fonctions et les tests, on remarque que le corps de l'itération est introduit par un deux-points et est convenablement indenté.
Pour afficher les éléments successifs d'un itérable, il est possible le code suivant :

```
L = [1,2,4]
for elt in L:
    print(elt)
```

Remarque 3 :

Cette méthode ne permet de modifier les éléments.

Exemple 2 :

Proposer un code permettant d'obtenir le nombre d'éléments d'une liste

```
n = 0
for c in liste:
    n += 1
```

Remarque 4 :

Le code ci-dessus reproduit la fonction `len`. Le résultat de cette fonction est en réalité immédiat et ne nécessite pas de reparcourir toute la liste. Python associe directement le nombre d'éléments à une liste

Exemple 3 :

Proposer un code permettant d'obtenir la somme d'une collection numérique.

```
s = 0
for elt in L:
    s += elt
```

Pour un dictionnaire, le parcours par itérateur renvoie les différentes clefs :

```
mes_fruits = {"poire": 3, "pomme": 4, "orange": 2}
for cle in mes_fruits:
    print (cle, mes_fruits[cle])
# poire 2 ...
```

b) La fonction range

La fonction `range(start,stop,step)` renvoie un itérable (de type range) un peu particulier qui permet de définir rapidement une progression arithmétique. La fonction range prend trois arguments entiers : `(start,stop,step)` énumère les entiers de `start` (inclus) à `stop` (exclus) avec un pas de `step`. Sans le troisième argument, `range(debut, fin)`, le pas est de 1 par défaut. Avec un seul argument, `range(fin)`, le début est 0 par défaut.

On peut combiner la fonction range() et la fonction len() pour accèder aux différentes valeurs d'une liste. On obtient alors

```
N=len(liste)
for k in range(N):# k prend les valeurs 0,1,...N-1
    print(liste[k])
```

Remarque 5 :

`range(N)` peut être considéré comme une liste de N éléments et k est à la fois l'itérateur de range(N) et l'index de la liste.

Sous Python, l'objet `range` est une collection générée à la volée. Il est possible de définir `range(100 000 000)` sans que cela ne prenne trop d'espace mémoire : les différents entiers sont créés les uns à la suite des autres sans que soit générée réellement une liste de 100 000 000 d'éléments.

Exemple 4 :

Réaliser un code effectuant la somme de deux listes de même taille.

```
s = 0
for k in range(len(L1)):
    s += L1[k] + L2[k]
```

Exemple 5 :

Définir la liste des distances entre le point P_0 et les autres. On utilisera la liste `points` écrit sous la forme :

$$points = [[x_0,y_0],[x_1,y_1],...]$$

4 - Opérations spécifiques sur les listes

a) Concaténation

On parle de concaténation lorsque l'on empile deux chaînes de caractères ensembles. Cette opération est réalisée à l'aide de l'opérateur "+".

```
>>> jourA = [ 'mercredi','jeudi']
>>> jourB = ['lundi','mardi']  # La présence des crochets est primordiale
>>> jour = jourB + jourA  # deux listes seulement peuvent être ajoutées
>>> print(jour)
'lundi', 'mardi', 'mercredi','jeudi'
```

b) Duplication

L'opérateur '*' pour les listes est un opérateur de *duplication*. Cet opérateur est couramment utilisé pour créer une liste d'éléments identiques.

```
L=4*[0]
-> L=[0,0,0,0]
```

c) Déconstructionn dépaquetage

L'opérateur '=' permet de l'affectation multiple de variables. Pour une liste de deux éléments : L = [0,3]

```
>>> x, y  = L
# x = 0 et y =3
```

Cela permet d'échanger rapidement le contenu de deux variables :

```
a = 1
b = 2
a,b = b,a
# a = 2 et b = 1
```

5 - Pour aller plus loin...

a) Allocation de la mémoire

Python est un langage orienté objet, des liens sont crées entre des cases mémoires contenant des valeurs et des cases mémoires contenant le nom des variables.
Pour des variables simples (float/int), les cases mémoires sont liés directement aux variables.

```
a=2   # a -> 2
b=a   # b -> 2
a=3   # a -> 3 mais b->2
```

Pour des listes, la modification d'un élément de la liste se répercute sur les autres objets pointant vers la même liste.

```
a=[2,3]   # a -> [2,3]
b=a       # b -> [2,3] même case mémoire
a[0]=1    # a -> [1,3] et b->[1,3]
```

Pour copier une liste d'une ancienne à une nouvelle variable, et éviter que toute modification de l'ancienne n'affecte la nouvelle, on utilisera l'instruction suivante :

```
a = [2,3]   # a -> [2,3]
b = a[:]    # b -> [2,3] sur une autre case mémoire
# ou encore b = list(a)
a[0] = 1    # a -> [1,3] et b-> [2,3]
```

b) Compréhension

C'est une idée reconnue mais peu de langages de programmation l'ont adoptée : la possibilité de créer des listes de manière à la fois concise et élégante. Profitons-en ! La syntaxe est très proche de la manière dont, en maths, on peut décrire certains ensembles :

$$\{f(x); x \in A\} et \{x \in A \mid P(x)\}$$

```
>>>[x for x in range(0, 10) if x\%2==0]
[0, 2, 4, 6, 8]
>>>[x**2 for x in range(0,4)]
[0, 1, 4, 9]
```

6 - Autres données itérables

a) Les chaînes de caractères

Définition :
Une chaine de caractères type 'Str', string, est une collection d'éléments ordonnés **non mutable** composée de lettres.

On accède aux éléments par indice :

```
mot = "MPSI"
mot[1] #P
mot[0]='P'  #Error
```

le nombre de caratères est donné par `len`. L'opérateur '+' est la concaténation.

```
mot = "MPSI"
mot= 'PC' +mot[2:4]
```

b) Les tuples

Définition :
Un tuple 'tuple', string, est une collection d'éléments ordonnés **non mutable** .

c) Les dictionnaires

Comme les listes, les dictionnaires permettent de "stocker" des données. Chaque élément d'un dictionnaire est composé de 2 parties, on parle de paires "clé/valeur". Voici un exemple de dictionnaire :

```
mon_dico = {"nom": "Durand", "prenom": "Christophe"}
```

Les dictionnaires utilisent des accolades pour définir le début et la fin du contenu alors que les listes utilisent des crochets [] et les tuples des parenthèses. Dans le dictionnaire ci-dessus, "nom", "prenom" sont des clés et "Durand", "Christophe" sont des

valeurs. La clé "nom" est associée à la valeur "Durand", la clé "prenom" est associée à la valeur "Christophe" et la clé "date de naissance" est associée à la valeur "29/02/1981". Les clés sont des chaînes de caractères ou des nombres. Les valeurs peuvent être des chaînes de caractères, des nombres, des booléens... Contrairement aux listes et tuples qui sont des collections ordonnées L[0], L[1]...., les

Propriété :
Il est facile d'ajouter un élément à un dictionnaire (les dictionnaires sont mutables)

Pour créer un dictionnaire, il est aussi possible de procéder comme suit :

```
mon_dico = {}
mon_dico["nom"] = "Durand"
mon_dico["prenom"] = "Christophe"
mon_dico["date de naissance"] = "29/02/1981"
```

d) Bilan des collections itérables

caractéristiques	liste	tuple	chaine de caractères	dictionnaire
accès	indice : L[0]	indice : t[0]	indice : c[0]	clef : dico[clef]
parcours	for elt in L	for elt in t	for carac in chaine	for cle in dico
mutation	oui	non	non	oui

Similitudes et différences

Les attributs usuels des listes, chaînes et tuple Les fonctions, opérations et méthodes ci-dessous sont à connaître

- nombre de l'éléments : `len`
- pour N éléments, accès entre 0 et N-1
- opérateur de concaténation : "+"
- opérateur de répétition : "*"

pour les listes uniquement :

- suppression et renvoie du dernier élément : méthode `pop()` (idem pour dictionnaire)
- ajout d'un élément : méthode `.append(elt)`

III. Introduction aux fonctions

1 - Présentation

Afin de simplifier l'écriture d'un code, on utilise des fonctions au sens informatique du terme.

Définition :
Une fonction (informatique) est un sous-programme qui effectue une tâche ou un calcul relativement indépendant du reste du programme.

Une fonction est un objet qui prend des valeurs en paramètres et, après calculs, renvoie un autre objet. En python, elle est définie par le mot clé `def` :

```
def ma_fonction(arg1,arg2,...):
    instructions
    instructions
    return valeur
```

Une fonction est donc caractérisée par

- Son nom `ma_fonction` ;
- Ses arguments (ou paramètres) `arg1, arg2` ;
- Son contenu (corps) ;
- Sa valeur de retour (éventuelle) après le mot clé `return`.

Exemple 6 :
Proposer l'écriture d'une fonction `longueur` d'argument `L` qui renvoie le nombre d'éléments de la liste `L`.

```
def longueur(L):
    c = 0
    for x in L :
        c += 1
    return c
```

Remarque 6 :
Cette fonction n'a aucun intérêt car la fonction `len` permet d'avoir sans calcul le nombre d'élément d'une liste.

2 - Les bons usages

Une fonction permet :

— de réutiliser plusieurs fois du code déjà écrit ;

— de faciliter la lecture du code source (petits programmes courts et clairs) ;

— de corriger plus simplement les bogues par des tests ;

— d'organiser et de structurer un projet.

L'écriture grâce aux fonctions permet d'effectuer des tests simples pour vérifier que le code écrit est correct. En TD, on essaiera d'effectuer les tests de la façon suivante :

```
1  def longueur(L):
2      c = 0
3      for x in L :
4          c += 1
5      return c
6  # TEST
7  L = []
8  print(longueur(L), ' =  0')
9  L = [1,2,3,4,5]
10 print(longueur(L), ' = 5')
```

IV. Structure conditionnelle

1 - Variable booléenne

Définition :
Une variable booléenne ne peut prendre que deux valeurs `True` ou `False`.

Elle est en général issue une proposition dont on peut dire si elle est vraie ou fausse. Il s'agit souvent d'une égalité ou d'une inégalité que Python évalue pour savoir si elle est vraie ou fausse.

Par exemple, si x vaut 3, la condition " $x > 0$ " est vraie. Par contre, si x vaut -1, la condition " $x > 0$ " est fausse. On peut affecter le résultat de cette comparaison à une variable :

```
1  bool = 2**10<10**3   #valeur : False
```

Chacun des opérateurs ci-dessous retourne des booléens.

Remarque 7 :
Il faut être vigilant sur le test d'égalité donné par l'opérateur '==' qu'il ne faut pas confondre avec l'opérateur d'affectation '='.

Opérateurs	Signification
< >	inférieur, supérieur
<=, >=	inférieur ou égal, supérieur ou égal
==	égal
!=	différent

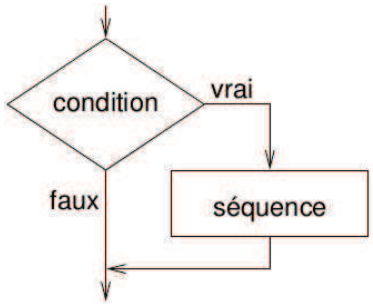
TABLE 1.1 – Opérateurs de comparaison

```
1  x = 5 <= 6 # True
2  y = 5 \\
3  y == 5   # True
4  y == 6   # False
```

2 - Structure conditionnelle simple

Pour construire des algorithmes, nous disposons d'instructions capables de tester une condition et de modifier le comportement du programme. Pour cela on utilise la commande **if**

Représentation schématique :



En code Python, la structure conditionnelle présente une indentation pour délimiter les instructions et ' :' à la fin de la condition

```
if conditions:
   instructions
```

L'exécution d'une telle commande se fait de la façon suivante : Si la condition *conditions1* est vraie alors on exécute les instructions *instructions1* sinon rien ne se passe.

Exemple 7 :

Proposer une fonction `occurence(chaine,lettre)` permettant de compter le nombre d'occurence d'une lettre dans un e chaîne de caractère.

```
def occurence(chaine,lettre):
    c = 0
    for caractere in chaine :
        if caractere == lettre :
            c+=1
    return c
```

Remarque 8 :

En cryptographie, le chiffrement par décalage, aussi connu comme le code de César, est une méthode de chiffrement très simple utilisée par Jules César dans ses correspondances secrètes. Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début. Il s'agit d'une permutation circulaire de l'alphabet. La longueur du décalage constitue la clé du chiffrement qu'il suffit de transmettre au destinataire. La fonction `occurence(chaine,lettre)` permet de retrouver cette clé de chiffrement en déterminant le caractère le plus redondant.

0

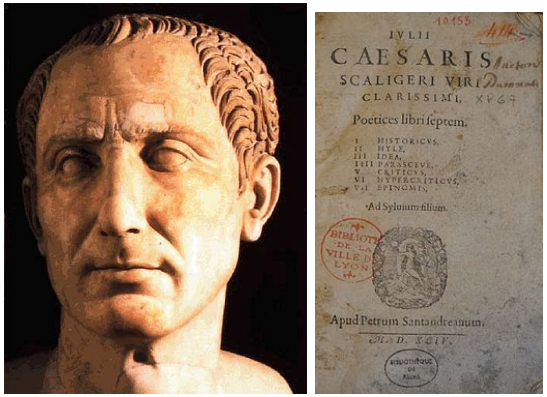


FIGURE 1.1 – Jules Cesar

3 - Les connecteurs logiques

Les connecteurs logiques lient des opérateurs booléens entre eux. Ils permettent ainsi d'évaluer le résultat final de plusieurs comparaisons.

L'utilisation des connecteurs logiques doit correspondre avec la table de vérité suivante :

Opérateurs	Signification
and	et logique
or	ou logique
not	négation logique

TABLE 1.2 – Connecteurs logiques

P	Q	Pet Q	P ou Q
F	F	F	F
F	V	F	V
V	F	F	V
V	V	V	V

TABLE 1.3 – table de vérité ET./OU

Exemple 8 :

Que renvoie les booléens suivantes :

```
y = 5 \\
y > 4 and y < 6
y > 4 or  y > 6
x = True and False
x = True or False
y=not x  -> False
```

En langage python, il est possible d'écrire des encadrements de valeurs numériques plutôt que d'utiliser des connecteurs logiques. Les deux booléens ci-dessous sont équivalents.

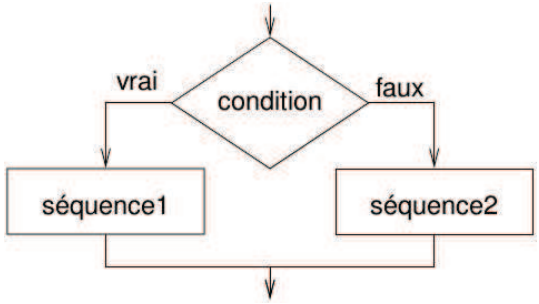
```
b1 = 3 < y and y < 4
b2 = 3 < y < 4
```

4 - La structure conditionnelle si-sinon

En code Python :

```
if conditions1:
    instructions1
else:
    instructions2
```

La représentation schématique est la suivante



L'exécution d'une telle commande se fait de la façon suivante : Si la condition *conditions1* est vraie alors on exécute les instructions *instructions1* sinon on exécute les instructions *instructions2* (lorsque ce bloc est présent avec un else)

Remarque 9 :

On n'oubliera pas les **deux points** à la fin de la ligne du if et après else : ("sinon" en anglais)

On s'assurera que la ligne après if est **indentée** (c'est-à-dire en retrait)

Exemple 9 :

Proposer une fonction `parite(L)` qui renvoie deux sous-listes issues d'une liste `L` numérique séparant les paires et les impaires

```
def parite(L):
    L_p, L_i= [] , []
    for x in L :
        if x%2==0:
            L_p.append(x)
        else :
            L_i.append(x)
    return L_p,L_i
```

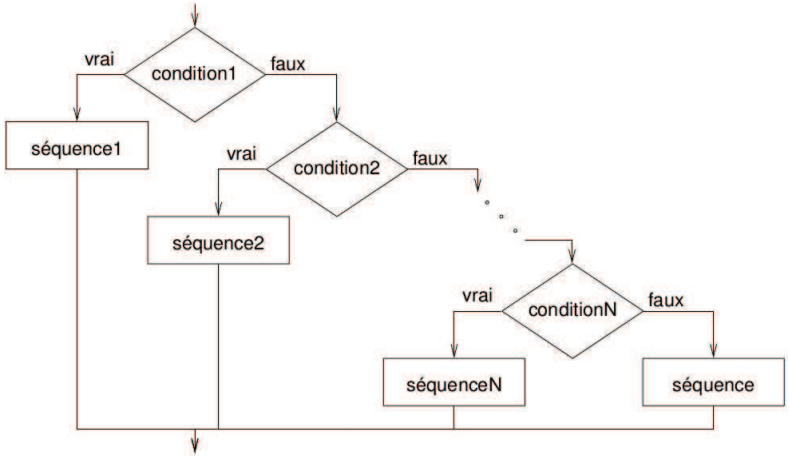
5 - Les structures conditionnelles si-sinon-si

Lorsque plusieurs cas apparaissent, on peut utiliser des structures conditionnelles emboîtées

En code Python

```
if conditions1:
    instructions1
elif conditions2:
    instructions2
.
.
.
else:
    instructionsn
```

Représentation schématique :



L'exécution d'une telle commande se fait de la façon suivante :
Si la condition *conditions1* est vraie alors on exécute les instructions *instructions1*.
Sinon si la condition *conditions2* est vraie alors on exécute les instructions *instructions2* etc...
Sinon (si aucune des conditions précédentes n'est vérifiée) on exécute les instructions *instructions*

On remarque que **elif** est la contraction de **else if**

Exemple 10 :

On considère une liste contenant une succession de prénoms et de moyennes de la forme `L= [ ["Julien",16.2], ["Romain",10.0] ,[...] ]` . Proposer une fonction `mention` renvoyant deux listes. La première composée d'une liste de listes composées des prenoms et des mentions, la seconde composée des personnes sans mention.

```
def mention(L):
    R1,R2 = [ ], [ ]
    for liste in L :
        prenom,note = liste
        if note >= 16:
            R1.append( [prenom, 'tres bien'])
        elif note >= 14:
            R1.append( [prenom, 'bien'])
        elif note >= 12:
            R1.append( [prenom, 'assez bien'])
        else :
            R2.append([prenom])
    return R1,R2
```

V. Ecriture d'une fonction

1 - Documentation

Une fonction peut prendre n'importe quel type d'argument, voire pas d'arguments du tout. Il est nécessaire de bien préciser les types de variables attendus dans une fonction. On précise alors le but de la fonction dans des commentaires.

Remarque 10 :

Le choix du nom des arguments permet de s'affranchir de commentaires lorsque les noms sont significatifs.Pour les noms de fonctions ou de méthodes : le nom est entièrement écrit en minuscules et les mots sont séparés par des signes soulignés (_). Exemple : `nom_de_fonction.`

```
def cos2(x):
    """
    x : float
    retour : float
    """
    return cos(x)**2
```

Définition :
L'utilisation d'annotations permet également de donner des indications sur la fonction utilisée en précisant le type de chaque variable.

```
def cos2(x : float) -> float:
    return cos(x)**2
```

2 - Fonctions usuelles

a) Fonctions à but mathématique

Une fonction peut avoir en argument une autre fonction.

Exemple 11 :
Quel est le but de la fonction suivante :

```
def mystere(f,x,h):
    return ( f(x+h) - f(x) ) / h
```

b) Caractéristiques d'une liste

Max, Min, Moyenne Variance
position d'un élément

c) Manipulation de plusieurs listes

Exemple 12 :
Proposer une fonction `distance` documentée d'arguments `P1 = (x1,y1)` et `P2 = (x2,y2)` renvoyant la distance entre ces points. La fonction `sqrt` est supposée importée.

```
def distance(P1,P2):
    """
    P1,P2 : tuple de coordonnees
    retour : distance (float)
    """
    x1,y1 = P1
    x2,y2 = P2
    return sqrt( ( x2 - x1 )**2 + ( y2-y1 )**2 )
```

On peut également proposer un type pour les variables ainsi que pour la valeur de retour

```
def distance(P1:tuple,P2:tuple) ->float :
    x1,y1 = P1
    x2,y2 = P2
    return sqrt( ( x2 - x1 )**2 + ( y2 - y1 )**2 )
```

Remarque 11 :
Les types primitifs sont les plus récurrents et utilisés : `bool`, `int`, `str`, `float`, `function`.

Somme terme à terme de deux listes

3 - Portée lexicale

a) Variable locale

Définition : *Variable locale*
Une variable locale est une variable déclarée et utilisée dans une fonction. Elle est accessible en lecture et en écriture uniquement dans cette fonction. Elle est à portée locale.

Dans le code ci-dessous, la variable est une variable locale à la fonction **f(x)**.

```
def f(x):
    a = 10
    return x+a
```

Si on souhaite modifier la valeur de a par exemple dans le programme principal, la variable a de la fonction ne sera pas modifiée car bien qu'elles portent le même nom, elles sont différentes du point de vue de l'interpréteur.

```
1 def f(x):
2     a = 10
3     return x+a
4 a=2
5 print(f(5)) # affichage : 15
```

Remarque 12 :

Pour modifier une variable locale, il est souhaitable de la mettre en paramètre :

```
1 def f(x,a):
2     return x+a
3 print(f(5,2)) # affichage : 7
```

b) Modèle LEG

Une fonction tente d'abord de récupérer une variable Localement.

Puis dans l'environnement Englobant la fonction puis Globalement. Dans le code ci-dessous, la variable a n'apparaît pas dans la fonction `f(x)` ci-dessous. Python va donc la chercher dans l'environnement " supérieur ", c'est à dire dans la partie du code présentant une indentation de moins.

```
1 def f(x):
2     return x+a
3 a = 5
4 print(f(3))
```

Propriété :

Il est impossible de modifier une variable située dans un environnement englobant.

L'exemple ci-dessous renvoie le message d'erreur `UnboundLocalError: local variable 'a' referenced before assignment`. La variable 'a' n'est pas définie pour la fonction f.

```
1 def f(x):
2     a = a+2
3     return x+a
4 a = 5
5 print(f(3))
```

c) Variable globale

Définition : *Variable globale*

Une variable globale est une variable déclarée avec le mot clé **global** soit dans une fonction ou soit dans le programme principal.
C'est une variable accessible en lecture et en écriture à la fois par le programme principal et le(s) fonction(s). Elle est à portée globale.

Par exemple, si on souhaite modifier la valeur de la variable a de la fonction **f(x)** depuis le programme principal, il faut déclarer cette dernière en global dans le programme principal ou dans la fonction.

```
1 def f(x):
2     global a
3     a += 2
4     return x+a
5 a=5
6 f(1)
7 print(a) # affichage : 7
```

Remarque 13 :

La modification de la valeur de la variable globale est mise à jour dans toutes les instructions qui l'utilisent. C'est une variable partagée. La variable a est déclarée et modifiée par le programme principal. La fonction **f(x)** utilise la même variable pour effectuer son calcul.

VI. Fonction et procédure...

1 - Description

Définition :
Dans le cas général, les fonctions peuvent opérer sur **tout type de variable** comme des valeurs numériques ou des chaînes de caractères afin de réaliser des traitements et/ou des calculs. Lorsqu'il n'y pas de valeur de retour, on parle de *procédure*.

Certaines fonctions usuelles sont des procédures, par exemple :

- **print(chaîne)** pour afficher des données dans la console Python :
- **write(fichier,chaîne)**, une fonction pour sauvegarder des données dans un fichier :

Remarque 14 :
La dénomination **procédure** tend à disparaître au profit du terme générique **fonction.**

2 - Mutation sur place

Propriété :
Les fonctions peuvent modifier des listes sans valeur de retour.

Bien qu'il s'agisse d'un paramètre, l'action d'une fonction est différente sur une liste par rapport à une variable.

```
1 def modifie(a):
2     a = 2
3 a = 3
4 modifie(a)
5 print(a)  # a = 3
```

Dans l'exemple proposé, 'a' est un argument mais est aussi une variable locale. Sa valeur n'est pas modifiée. En revanche ...

```
1 def modifie(L):
2     L[2] = 4
3 L = [1,2,3]
4 modifie(L)
5 print(L)  # L = [1,2,4]
```

Une liste peut être modifiée car la fonction va modifier les pointeurs vers chaque élément de la liste. Ce phénomène s'explique par la différence de principe pour le stockage des informations. Une variable numérique 'a' est directement associée par un pointeur vers une case mémoire associée à une valeur numérique. Par contre, une liste 'L' est associée à un ensemble de pointeurs vers différentes cases mémoires. Une fonction peut donc modifier les pointeurs qui ne sont pas considérées comme des variables locales.

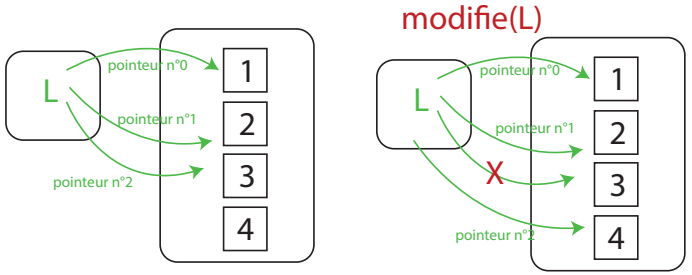


FIGURE 1.2 – Illustration de la modification des pointeurs

Exemple 13 :
On considère une liste constituée de prénoms et des notes du dernier devoir : `L= [ ['Nathan',15] , ['Thibaut',9],... ]` . Proposer une fonction `supp_majorant` supprimant le majorant. La méthode `.remove(elt)` permet de supprimer l'élément `elt` à une liste L.

```
1 def supp_majorant(L):
2     majorant = L[0]
3     for eleve in L:
4         prenom,note = eleve
5         if note > majorant[1] :
6             majorant = eleve
7     L.remove(majorant)
```

Définition :

Lorsque la fonction ne présente pas de valeur de retour mais modifie le contenue d'une liste `L`. On dit que la mutation de la liste `L` s'effectue **en place** ou **sur place**.

Exemple 14 :

On considère la fonction ci-dessous.

```
1  def mystere(L):
2      L2 = []
3      for k in range(len(L)):
4          L2.append(L.pop())
5      return L2
```

☐ 1 - Quel est le but de la fonction mystère ?

☐ 2 - Si `L=[3,4,5]`, que devient `L` une fois la fonction exécutée

☐ 3 - Proposer une modification du code pour éviter que `L` ne soit modifiée.

1. La fonction `mystere` renvoie une liste inversée.
2. Une fois la fonction exécutée, la liste `L` est vide.
3. Pour éviter la desctruction de `L` on peut travailler sur une copie.

```
1  def mystere(L):
2      L2 = []
3      L_copy = L[:]
4      for k in range(len(L)):
5          L2.append(L_copy.pop())
6      return L2
```