Introduction à la Programmation en Python

Cours et Travaux Dirigés — Deuxième Partie

Licence
Université de Paris
Année académique 2020–2021

Introduction à la Programmation 1 PYTHON

MT11Y060

Séance 6 de cours/TD

Université de Paris

Objectifs:

- Boucles `while`.
- Variables booléennes.

1 La boucle “while”

Boucle non bornée [COURS]

La boucle non bornée permet de répéter des instructions tant qu'une expression booléenne s'évalue à `True`. Elle est utile lorsqu'on ne connaît pas *a priori* le nombre d'itérations nécessaires.

```
while (expression booléenne) :
    instructions à répéter
```

- L'expression booléenne est appelée condition ou encore test d'arrêt de la boucle.
- Par exemple, la boucle suivante s'arrêtera quand la valeur de la variable entière `a` sera 0 après avoir affiché 50 lignes contenant la chaîne « `Hello` ».

```
a = 50
while (a > 0) :
    print ("Hello")
    a = a - 1
```

- Une boucle non bornée peut ne jamais terminer si sa condition est toujours vraie. La plupart du temps, la non terminaison du programme n'est pas le comportement escompté car on souhaite obtenir un résultat ! (Certains programmes sont des exceptions à cette remarque. Par exemple les serveurs WEB doivent néanmoins ne jamais s'arrêter).
- Par exemple, la boucle suivante ne termine jamais et l'instruction « `print ("Bonjour")` » n'est donc jamais exécutée :

```
b = 0
while (b == 0) :
    b = b // 2
print ("Bonjour")
```

Exercice 1 (Première boucle, $\star$)

Quelle est la valeur de la variable `r` *à la fin de la suite des instructions suivantes ?*

```
n = 49
r = 0
```

```
while (r * r < n) :
    r = r + 1
```

□

Exercice 2 (Les "for"s vus comme des "while"s, ⋆)

Réécrivez la suite d'instructions suivante pour obtenir le même comportement en utilisant un "`while`" à la place du "`for`".

```
a = 0
for i in range (0, 32, 1) :
    a = a + i
print (a)
```

□

Exercice 3 (Terminaison, ⋆)

Qu'affichent les deux séquences d'instructions suivantes ? Est-ce que leur exécution se termine ?

```
n = 2
while (n > 0) :
    print (n)
```

```
n = 2
while (n > 0) :
    n = n * 2
    print (n)
```

□

Exercice 4 (L'ordinateur obstiné, ⋆)
Écrire un programme qui lit un entier du clavier (en utilisant `input()`). Tant que la valeur entrée est impaire demander à l'utilisateur une nouvelle valeur. Quand la valeur entrée est paire, le programme doit l'afficher et terminer. □

2 Variables de type `bool`

Présentation du type `bool` [Cours]

— Comme toute valeur, une valeur de type `bool` (qui correspond à un booléen) peut être stockée dans une variable.
— Rappel : Il y a deux valeurs pour le type `bool` qui sont `True` et `False`.
— Par exemple, on peut initialiser une variable booléenne ainsi :

```
b = False
```

ou bien encore comme cela :

```
b = x >5
```

Dans ce dernier cas, si la valeur contenue dans `x` est strictement plus grande que 5 alors l'expression booléenne « `x > 5` » s'évaluera en la valeur `True`, qui sera la valeur initiale de la variable `b`.

— Rappels : les opérateurs booléens sont `or` (**ou**), `and` (**et**) et `not` (**non**).

— Par exemple, les affectations suivantes sont valides :

```
b1 = False #b1 vaut False.
b2 = (3 + 2 < 6) # b2 vaut True.
b1 = not (b2)  # b1 vaut la negation de True, donc False.
b2 = b1 or b2 # b2 vaut False or True, donc True.
```

— **Attention :** le symbole = est utilisé dans les instructions d'affectation tandis que le symbole == est un opérateur de test d'égalité entre entiers ou entre chaînes de caractères.

— Comme toute valeur, une valeur de type `bool` peut être passée en paramètre à une fonction ou une procédure et renvoyée par une fonction.

Exercice 5 (Évaluer, ⋆)

Quelles sont les valeurs des variables `x`, `b`, `d`, `e` *et* `f` *après avoir exécuté les instructions suivantes ?*

```
x = 3 + 5
b = (x == 5 + 3)
d = (x > x + 1)
e = b or d
f = b and d
f = (e != b)
```

□

Exercice 6 (Parité, ⋆)

Écrire une fonction « `isEven (a)` *» qui prend en paramètre un entier et renvoie* `True` *si cet entier est pair,* `False` *sinon. Savez-vous écrire cette fonction sans de vous servir de l'instruction* `if` *?* □

Exemples d'utilisation de variables booléennes [Cours]

— Les variables booléennes peuvent être utilisées pour signaler qu'une condition a été vérifiée, ce qui peut être utile pour arrêter une boucle.

— On peut aussi utiliser les variables booléennes comme paramètres de fonction pour faire varier le comportement selon des conditions.

— Les variables booléennes sont parfois appelées drapeaux (*flag* en anglais).

Exercice 7 (Drapeau et recherche, ⋆)

1. *On considère la suite d'instructions suivante qui parcourt une liste* `l` *d'entiers et affecte* `True` *à la variable* `found` *si une des cases de* `l` *vaut* 3 *:*

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
i = 0
found = False  # flag
while (i < len (l)) :
  if (l[i] == 3) :
    found = True
  i = i + 1
```

   Quelle est la valeur contenue dans la variable `i` *après ces instructions ?*

2. *Même question sur la suite d'instructions suivante :*

```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
i = 0
found = False  # flag
```

```
while (i < len (l) and not (found)) :
  if (l[i] == 3) :
    found = True
  i = i + 1
```

3. *Qu'en déduisez-vous ?*
4. *Écrire une fonction « `findValueList (l, v)` » qui teste si la valeur contenue dans `v` est présente dans la liste d'entiers `l`.*

□

Exercice 8 (Paramètre booléen, ⋆)
Écrire une procédure qui énumère les entiers entre 1 et n. La procédure prend deux paramètres : l'entier n, et un booléen `up`. Si le paramètre booléen est vrai, on affiche les entiers de 1 à n, sinon on affiche les entiers de n à 1. □

3 DIY

Exercice 9 (Affichage maîtrisé, ⋆)
Écrire, à l'aide d'une boucle, un programme qui affiche la chaîne `"bla"` plusieurs fois de suite autant que possible, sans dépasser les 80 caractères (longueur standard d'une ligne dans un terminal).

□

Exercice 10 (Comptine, ⋆)
Modifier les instructions suivantes pour que l'accord de kilomètre(s) soit correct.

```
n = 10
while (n > 0) :
    print (n, end="")
    print (" kilometre a pied ca use les souliers")
    n = n - 1
```

□

Exercice 11 (Palindrome, ⋆⋆)
Un mot est un palindrome si on obtient le même mot en le lisant à l'envers. Par exemple "kayak", "ressasser", ou "laval" sont des palindromes. Écrire une fonction qui prend en paramètre une chaîne de caractères `s` et qui renvoie le booléen `True` si le mot `s` est un palindrome et `False` sinon. □

Exercice 12 (Logarithme, ⋆)
Écrire une fonction qui prend en paramètre un entier n, et renvoie le nombre de fois qu'il faut diviser celui-ci par deux avant que le résultat soit inférieur ou égal à 1. □

Exercice 13 (n-ème chiffre, ⋆⋆)
Écrire une fonction qui prend en paramètres deux entiers k et n et renvoie le n-ème chiffre de k, ou 0 si k n'est pas aussi long (par exemple le 2ème chiffre de 1789 est 8). □

Exercice 14 (Binaire, ⋆⋆⋆)
Écrire une fonction qui prend en paramètre un entier `n` et qui renvoie la représentation binaire de `n` sous la forme d'une chaîne de caractères formée de caractères 0 et 1. □

Exercice 15 (lastOcc, ⋆⋆)
Écrire une fonction « `lastOcc(li, x)` » qui prend en paramètre une liste d'entiers et une valeur entière. Si la liste contient cette valeur, la fonction renvoie l'indice de la dernière occurrence de la valeur. Si la liste ne contient pas cette valeur, la fonction renvoie -1.

Cet exercice a déjà été traité au TD 4, mais on demande de remplacer la boucle `for` par une boucle `while`. □

Exercice 16 (Syracuse, ⋆⋆⋆)
La suite de Syracuse de premier terme p (entier strictement positif) est définie par $a_0 = p$ et

$$a_{n+1} = \begin{cases} \frac{a_n}{2} & \text{si } a_n \text{ est pair} \\ 3 \cdot a_n + 1 & \text{si } a_n \text{ est impair} \end{cases} \quad \text{pour } n \geq 0$$

Une conjecture (jamais prouvée à ce jour !) est que toute suite de Syracuse contient un terme $a_m = 1$. Trouver le premier m pour lequel cela arrive, étant donné p. □

Exercice 17 (Itérations, ⋆)
Combien y a-t-il d'itérations dans la boucle suivante :

```
n = 15
while (n >= 0) :
    n = n - 1
```

Même question pour la boucle suivante, en fonction du paramètre `n` *:*

```
def nbIterations (n) :
    while (n >= 0) :
        n = n - 1
```

□

Exercice 18 (Logarithme (itéré), ⋆⋆)

1. *Le logarithme en base 2 d'un entier $n \geq 1$ (noté $\log_2 n$) est le réel x tel que $2^x = n$. On se propose de calculer la partie entière de $\log_2 n$, c'est-à-dire le plus grand entier m tel que $2^m \leq n$. On notera l la partie entière du logarithme en base 2, c'est-à-dire que $m = l(n)$. Par exemple, $l(8) = l(9) = 3$ car $2^3 = 8 \leq 9 < 2^4$.*
   Écrire une fonction `lo` *qui prend en paramètre un entier n et renvoie la partie entière* `lo (n)` *de son logarithme en base 2. On calculera* `lo (n)` *en effectuant des divisions successives par 2.*
2. *À partir de la fonction* `lo` *précédente, on peut définir une fonction* `loStar` *ainsi :* `loStar (n)` *est le plus petit entier i tel que la i-ème itération de* `lo` *sur l'entrée n vaille 0. Par exemple,* `loStar (1)` *vaut 1 car dès la première itération,* `lo (1)` *vaut 0 ; ou encore* `loStar (2)` *vaut 2 car* `lo (2)` *vaut 1 et* `lo (1)` *vaut 0. Pour prendre un exemple plus grand, on a* `loStar (1500)` *vaut 4 car* `lo (1500)` *vaut 10,* `lo (10)` *vaut 3,* `lo (3)` *vaut 1 et* `lo (1)` *vaut 0 ; la quatrième itération de* `lo` *sur 1500 vaut donc 0.*
   Écrire la fonction `loStar` *qui prend en paramètre un entier n.*

□

Exercice 19 (Racine cubique, ⋆⋆)
Écrire une fonction qui renvoie la racine cubique d'un entier x, c'est-à-dire le plus petit entier a tel que $a^3 \geq x$. On peut s'inspirer de l'exercice 1. □

Exercice 20 (Racine $n$ième, ⋆⋆⋆)
Maintenant, écrire une fonction qui renvoie la racine $n$ième d'un entier x, c'est-à-dire le plus petit entier a tel que $a^n \geq x$. □

Introduction à la Programmation 1 PYTHON

MT11Y060

Séance 7 de cours/TD

Université de Paris

Objectifs:

- Connaître ses classiques sur les listes.
- Apprendre à voir quand un programme est faux.

1 Devenir incollable sur les listes

Opérations à savoir faire sur les listes [COURS]

- **Rechercher**
  - Dire si un élément est présent dans une liste
  - Renvoyer la position d'un élément dans la liste (la première ou la dernière)
  - Compter le nombre de fois qu'un élément est présent dans une liste.
  - Savoir construire une liste des positions d'un élément dans une liste
- **Modifier**
  - Savoir échanger deux éléments de place dans une liste
  - Savoir renverser une liste
  - Savoir décaler tous les éléments d'une liste
  - Savoir appliquer une fonction à tous les éléments d'une liste

Exercice 1 (Présence dans une liste, ⋆)
On considère la fonction `isPresent (el, li)` *qui renvoie* `True` *si* `el` *est dans la liste* `li`.

```
def isPresent (el, li) :
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            return (True)
        else :
            return (False)
    return (False)

def isPresent (el, li) :
    b = True
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            b = True
        else :
            b = False
    return (b)

def isPresent (el, li) :
```

```
    b = True
    for i in range (0, len(li), 1) :
        return (li[i] == el)

def isPresent (el, li) :
    b = False
    i = 0
    while ( i < len (li) and b) :
        if (li[i] == el) :
            b = True
    return (b)

def isPresent (el, li) :
    b = False
    i = 0
    while ( i < len (li) and not(b)) :
        b = b and (li[i] == el)
    return (b)
```

code/exoPresent.py

1. *Pourquoi les propositions précédentes sont-elles fausses ? À chaque cas, donnez un exemple de listes d'entiers et d'éléments pour laquelle la fonction renvoie un mauvais résultat.*
2. *Proposez une correction pour chacun des cas.*

□

Exercice 2 (Occurences dans une liste, ⋆)
On considère la fonction `occ (el, li)` *qui renvoie le premier indice de* `el` *dans la liste* `li` *et* `-1` *si* `el` *n'est pas dans la liste.*

```
def occ (el, li) :
    for i in range (0, len(li), 1) :
        if (el == li[i]) :
            return (i)
        else :
            return (-1)
    return (-1)

def occ (el, li) :
    for i in range (0, len(li), 1) :
        if (el == li[i]) :
            return (i)
    return (i)

def occ (el, li) :
    p = 0
    for i in range (0, len(li), 1) :
        if (el == li[i]) :
            p = i
        else :
            p = -1
    return (p)

def occ (el, li) :
    p = 0
    for i in range (0, len(li), 1) :
        if (el == li[i]) :
```

```
            p = i
    return (p)

def occ (el, li) :
    p = -1
    i = 0
    while ( i < len(li)) :
        if (el == li [i]) :
            p = i
            i = i + 1
    return (p)

def occ (el, li) :
    p = -1
    i = 0
    while ( i < len(li) and p != -1 ) :
        if (el == li [i]) :
            p = i
        i = i + 1
    return (p)
```

code/exoOccurence.py

.

1. *Pourquoi les propositions précédentes sont-elles fausses ? À chaque cas, donnez un exemple de listes d'entiers et d'éléments pour laquelle la fonction renvoie un mauvais résultat.*
2. *Proposez une correction pour chacun des cas.*

□

Exercice 3 (Compter dans une liste, ⋆)

On considère la fonction `count (el, li)` *qui renvoie le nombre de fois que l'élément* `el` *apparaît dans la liste* `li`.

```
def count (el, li) :
    c = 0
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            c = c + 1
            return (c)
    return (c)

def count (el, li) :
    c = 0
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            c = c + 1
        else :
            c = c - 1
    return (c)
```

code/exoCounting.py

.

1. *Pourquoi les propositions précédentes sont-elles fausses ? À chaque cas, donnez un exemple de listes d'entiers et d'éléments pour laquelle la fonction renvoie un mauvais résultat.*
2. *Proposez une correction pour chacun des cas.*

□

Exercice 4 (Construire la liste des positions, ⋆)
On considère la fonction `pos (el, li)` *qui renvoie une liste contenant les positions où* `el` *apparaît dans la liste* `li`.

1. *Que valent les variables* `li1`, `li2` *et* `li3` *après exécution du programme suivant (en supposant que la fonction* `pos` *est correctement définie).*

```
li1 = pos (2, [3, 4, 5, 8])
li2 = pos (3, [3, 3, 4, 3, 4, 5, 6, 4, 3])
li3 = pos (4, [3, 3, 4, 3, 4, 5, 6, 4, 3])
```

2. *On considère les propositions suivantes pour la fonction* `pos (el, li)` *(on suppose que la fonction* `count` *est celle de l'exercice précédent et qu'elle est correcte).*

```
def count(el,li):
    c = 0
    for i in range(0,len(li),1):
        if el == li[i]:
            c = c+1
    return(c)

def pos1 (el, li) :
    lret = []
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            lret = [i]
    return lret

def pos2 (el, li) :
    n = count (el, li)
    lret = [0] * n
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            lret[i] = i
    return lret


def pos3 (el, li) :
    n = count (el, li)
    lret = [0] * n
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            lret[el] = i
    return lret

def pos4 (el, li) :
    n = count (el, li)
    lret = [0] * n
    j = 0
    for i in range (0, len(li), 1) :
        if (li[i] == el) :
            lret[j] = i
    return lret

l = [0,1,0,2,0,3]
print(pos4(0,l))
```

code/exoPos.py

a *Pourquoi les propositions précédentes sont-elles fausses ? À chaque cas, donnez un exemple de listes d'entiers et d'éléments pour laquelle la fonction renvoie un mauvais résultat.*

b *Proposez une correction.*

□

Exercice 5 (Échanger deux éléments dans une liste, ⋆)

Dans cet exercice, on considère la fonction `swap (i , j, li)` *qui échange les valeurs des indices* `i` *et* `j` *de la liste* `li`.

1. *Pour comprendre ce qui se passe tout en s'échauffant dites ce qu'affiche le programme suivant :*

```
def modif (l) :
    l[0] = 5

li = [1, 2, 4, 8, 16, 31]
modif (li)
print (li)
```

code/exoSwap1.py

2. *En déduire ce que doit renvoyer la fonction* `swap`.
3. *Pourquoi la définition suivante de* `swap` *n'est pas correcte ? Proposez une correction.*

```
def swap (i , j, li) :
    if (i < len (li) and j < len(li)) :
        li[i] = li[j]
        li[j] = li[i]
```

code/exoSwap2.py

□

Exercice 6 (Inverser une liste d'éléments, ⋆)

Dans cet exercice, on considère la fonction `reverse (li)` *qui prend en paramètre une liste* `li` *et renvoie la liste où l'ordre des valeurs est inversée.*

1. *Que renvoie donc* `reverse ([0, 1, 2, 3, 4, 5])` *?*

```
def reverse (li) :
    for i in range (0, len (li), 1) :
        li[i] = li [len(li) - 1 - i]
    return li

def reverse (li) :
    for i in range (0, len (li), 1) :
        temp = li[i]
        li[i] = li [len(li) - 1 - i]
        li [len - 1 - i] = temp
    return li

def reverse (li) :
    l = []
    for i in range (0, len (li), 1) :
        l [i] = li [len(li) - 1 - i]
    return li
```

code/exoReverse.py

2. a *Pourquoi les propositions précédentes sont-elles fausses ? À chaque cas, donnez un exemple de listes d'entiers pour laquelle la fonction renvoie un mauvais résultat.*

   b *Proposez une correction.*

   c *Parmi les corrections proposées, quelles sont celles qui changent la liste* `li` *passée en paramètre et quelles sont celles qui la laissent inchangée ?*

□

Exercice 7 (Décaler les éléments d'une liste, ⋆⋆)
Dans cet exercice, on considère la fonction `shift (li, p)` *qui prend en paramètre une liste* `li` *et modifie cette liste en décalant (de façon cyclique vers la droite) ses éléments de* `p` *positions (en supposant que* `p` *est positif).*

1. *Que valent les variables* `li1`, `li2` *et* `li3` *après exécution du programme suivant (en supposant que la fonction* `shift` *est correctement définie).*

```
li1 = [3, 4, 5, 8]
shift (li1, 1)
li2 = [3, 3, 4, 3, 4, 5]
shift (li2, 2 )
li3 = [1, 2, 3, 4, 5, 6]
shift (li3, 7)
```

```
def shift1 (li, p) :
    for i in range (0, len (li), 1) :
        li [i] = li [(i - p) % len(li)]

def shift2 (li, p) :
    for i in range (0, len (li), 1) :
        temp = li [i]
        li [i - p] = temp

def shift3 (li, p) :
    for i in range (0, len (li), 1) :
        li [(i + p) % len(li)] = li [i]
```

code/exoShift.py

2. a *Pourquoi les propositions précédentes sont-elles fausses ? À chaque cas, donnez un exemple de listes d'entiers et d'éléments pour laquelle la fonction renvoie un mauvais résultat.*

   b *Proposez une solution. On pourra pour cela commencer par écrire une fonction* `shift1` *qui décale la liste d'un élément vers la droite et utiliser ensuite cette fonction pour faire* `shift`.

□

2 DIY

Pour certains exercices, un "contrat" est proposé : ce sont des tests que nous vous fournissons pour vérifier votre réponse. Si votre réponse ne passe pas ces tests, il y a une erreur ; si votre réponse passe les tests, rien n'est garanti, mais c'est vraisemblablement proche de la réponse. Quand aucun contrat n'est spécifié, à vous de trouver des tests à faire.

Exercice 8 (Primalité, ⋆⋆)
Un entier p est premier si $p \geq 2$ et s'il n'a pas d'autres diviseurs que 1 et lui-même.

1. *Écrire une fonction* `prime` *qui prend en paramètre un entier n et qui renvoie* `True` *si n est premier, ou* `False` *sinon.*
2. *Écrire une fonction* `next` *qui prend en entrée un entier x et qui renvoie le plus petit nombre premier $p \geq x$. On pourra bien sûr se servir de la fonction* `prime` *précédente.*
3. *Écrire une fonction* `number` *qui prend en entrée un entier y et qui renvoie le nombre de nombres premiers $p \leq y$. On pourra bien sûr se servir de la fonction* `prime`.

Contrat:
Pour la fonction `next` *:*

x = 2 $\rightarrow$ *2*
x = 10 $\rightarrow$ *11*
x = 20 $\rightarrow$ *23*

Pour la fonction `number` *:*

x = 10 $\rightarrow$ *4*
x = 20 $\rightarrow$ *8*

□

Exercice 9 (Multiplication de matrices, $\star\star$)
Dans cet exercice, on suppose qu'une matrice A à m lignes et n colonnes est encodée par une liste de listes contenant m listes de taille n. Par exemple, la matrice

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

est encodée en Python par `[[1, 2, 3], [4, 5, 6]]`.

Étant données deux matrices A (à m lignes et n colonnes) et B (à n lignes et p colonnes), le produit matriciel de A par B est une matrice C à m lignes et p colonnes telle que pour tout $1 \leq i \leq m$ et pour tout $1 \leq j \leq p$, le coefficient $C_{i,j}$ de C se trouvant à la i-ème ligne et la j-ième colonne est égal à :

$$C_{i,j} = \Sigma_{1 \leq k \leq n} A_{i,k} B_{k,j}$$

1. *Écrire une fonction* `isMatrix (A)` *qui prend en paramètre une liste de listes d'entiers et vérifie qu'il s'agit de l'encodage d'une matrice (c'est à dire que chaque sous-liste à la même longueur) ; elle renvoie* `True` *dans ce cas et* `False` *sinon.*
2. *Écrire une fonction* `nbLines (A)` *qui prend en paramètre une liste de listes d'entiers qui encode une matrice et renvoie son nombre de lignes.*
3. *Écrire une fonction* `nbColumns (A)` *qui prend en paramètre une liste de listes d'entiers qui encode une matrice et renvoie son nombre de colonnes.*
4. *Écrire une fonction* `matriProx (A, B)` *qui prend en paramètre deux listes de listes d'entiers, vérifie qu'il s'agit de deux matrices, et vérifie que le nombre de colonnes de* `A` *est égal au nombre de lignes de* `B`*. Si ces conditions ne sont pas satisfaites, la fonction renvoie* `[]` *et sinon elle renvoie une liste de listes contenant la matrice correspondant au produit matriciel de* `A` *par* `B`*.*

□

Exercice 10 (Addition, $\star\star\star$)
Le but de cet exercice est de programmer l'addition décimale. Les deux nombres à additionner sont donnés sous forme de listes.

Par exemple, `x = [7, 4, 3]` *et* `y = [1, 9]`*, dont la somme doit être retournée sous forme de liste, dans cet exemple,* `[7, 6, 2]`*.*

Écrire une fonction `add` *qui prend en paramètre deux listes et fait l'addition des deux nombres représentés par les listes.*

Par exemple, si les entrées sont les listes `t1 = [3, 4, 7], t2 =[9, 1]`*, cela représente la somme $743+19$. On calcule d'abord $9+3$ ce qui fait 2 avec une retenue de 1. Puis on calcule $4+1$, plus la retenue, ce qui donne 6, avec une retenue de 0. Enfin, le dernier chiffre est 7. À la fin, on doit renvoyer la liste* `[0, 7, 6, 2]`*.*
Remarque : *Le premier chiffre dans la liste (celui le plus à gauche) peut-être 0.*

Contrat:

`t1 = [2, 4, 3, 5]` `t2 = [4, 3, 6]` $\rightarrow$ *renvoie :* `[0, 2, 8, 7, 1]`
`t1 = [1, 2]` `t2 = [1, 3]` $\rightarrow$ *renvoie :* `[0, 2, 5]`
`t1 = [6, 1, 2]` `t2 = [0, 6, 3, 0]` $\rightarrow$ *renvoie :* `[1, 2, 4, 2]`

□

Exercice 11 (Pascal, $\star\star\star$)
Écrire un programme qui affiche le triangle de Pascal jusqu'au rang n où n est un entier demandé à l'utilisateur. Si l'utilisateur rentre 4, le programme affichera :

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
```

Le programme doit faire appel à deux fonctions auxiliaires la première qui, étant donné n, renvoie le triangle de Pascal sous forme d'une liste de listes d'entiers de n + 1 lignes, la i-ème ligne contenant les coefficients de la i-ème puissance du binôme $(a + b)$ et la deuxième qui sert à afficher une liste de liste en affichant sur chaque ligne le contenu de chacune des listes (en séparant les différentes valeurs par des espaces). □

Introduction à la Programmation 1 PYTHON

MT11Y060

Séance 8 de cours/TD

Université de Paris

Objectifs:

- Variables locales et globales
- Tester une fonction
- Décrire (spécifier) le comportement d'une fonction.
- Réaliser une spécification.

1 Approfondissements sur les fonctions

Rappels [COURS]

- Les fonctions et procédures peuvent prendre toutes sortes de *paramètres* : des entiers, des booléens, des couples, des listes, mais aussi des fonctions.

```
#La fonction eval evalue f sur inp
def eval (f, inp):
    return f(inp)
```

  En général, on écrit des fonctions qui prennent peu de paramètres, et on évite les paramètres redondants ou inutiles.
- Une fonction permet de *factoriser du code*. Par exemple, si l'on veut remplacer une lettre par une autre dans plusieurs chaînes de caractères, il est utile de créer une fonction que l'on puisse appliquer à chacune des chaînes plutôt que de faire des copier-coller comme dans le programme suivant :

```
s = "abracadabra"
r=""
for i in range(0, len(s), 1):
    if s[i]=='a':
        r = r + 'e'
    else:
        r = r + s[i]
print(r)

s = "magie"
r=""
for i in range(0, len(s), 1):
    if s[i]=='a':
        r = r + 'e'
    else:
        r = r + s[i]
print(r)
```

— Introduire des fonctions a aussi d'autres buts : en donnant des noms bien choisis aux fonctions et à leurs arguments, et en leur assignant des tâches clairement identifiées, on améliore la lisibilité du code, et sa réutilisabilité dans d'autres programmes.
— Une fonction *doit* renvoyer un résultat à l'aide de l'instruction `return`. Cette instruction stoppe l'exécution d'une *fonction* et termine immédiatement la fonction. Il faut faire attention lorsqu'on l'utilise à ne pas stopper l'exécution trop tôt, notamment en choisissant bien l'*indentation* dans une conditionnelle ou une boucle.

```
def replace_ab(s):
    r=""
    for i in range(0, len(s), 1):
        if (s[i]=='a'):
            r = r+"b"
        else:
            r = r+s[i]
        return r
```

Dans cet exemple, le `return` étant placé dans la boucle, l'exécution de la fonction s'arrête après le premier passage dans la boucle.

Exercice 1 (Bien concevoir ses fonctions, ⋆)
Critiquez la façon dont est conçu ce code et proposez une version améliorée.

```
def f23(t, n):
  z = 0
  for i in range(0, n, 1):
      z = z + t[i]
  print("Somme :"+str(z))

def f24(n, t, msg):
  h = 0
  for i in range(0, n, 1):
      h = h + t[i]
  return (h / n)

t = [1,2,15,30,2]

f23(t, len(t))
print("Moyenne : " + str(f24(t, len(t), "Moyenne")))
```

code/exoConception.py

□

Exercice 2 (Bien utiliser return, ⋆)

1. *Écrivez une fonction* `membership` *qui prend en argument un entier* `n` *et une liste d'entiers* `t`*, et qui renvoie* `True` *si* `n` *apparaît dans* `t`*, autrement dit si* `n=t[i]` *pour au moins un* `i`*. On cherchera à renvoyer le résultat le plus tôt possible.*
2. *En utilisant la fonction* `membership`*, écrivez une fonction* `inclusion` *qui prend en argument deux listes d'entiers* `t` *et* `u` *et qui renvoie* `True` *si l'ensemble des entiers de* `t` *est inclus dans l'ensemble des entiers de* `u`*, autrement dit,* `inclusion(t,u)` *renvoie* `True` *si pour tout indice* `i` *de* `t`*, l'entier* `t[i]` *apparait dans* `u`*.*

□

Variables locales, variables globales [Cours]

— Une variable définie dans le corps d'une fonction ou d'une procédure est **locale** à cette fonction. Les paramètres sont assimilés à des variables locales.
Une variable définie en dehors de toute fonction ou procédure est dite **globale**.
Par exemple, dans le programme ci-dessous,

```
flag = False
def f(x):
    if x>=3:
        a = 3
    else:
        a = x
    return x
flag = (f(5) == 3)
```

il y a 1 variable *globale* : `flag` et 2 variables *locales* à la fonction `f` : le paramètre `x` et la variable `a`.

— Les paramètres et variables locales d'une fonction ou d'une procédure n'appartiennent qu'à cette fonction, c'est la règle de *localité*. En particulier, une fonction (par exemple g) ne peut pas faire référence à une variable locale d'une autre fonction (par exemple la variable `a` de `f`). Les variables locales sont détruites après chaque exécution d'une fonction.

```
a = 3
def f(x):
    a = 0
    return a*x
print(f(2))
print(a)

```

Le programme ci-dessus affiche successivement 0 puis 3. Il est équivalent au programme suivant :

```
a = 3
def f(x):
    a_f = 0
    return a_f*x
print(f(2))
print(a)

```

— Au contraire, une variable globale est accessible par toutes les fonctions. Par défaut en Python, une variable globale peut être lue par une fonction ou une procédure. Si on veut pouvoir la modifier, il faut la déclarer `global`[1] au début de la fonction ou de la procédure.

```
count = 0
def show():
    print(count)
def inc():
    global count
    count = count+1
while count <5:
    inc()
show()

```

Dans le programme ci-dessus, la procédure `show` accède en lecture à la variable globale `count`, la procédure `inc` modifie la variable globale `count`.

— Quand on passe une liste en paramètre à une fonction ou à une procédure, ce n'est pas son contenu, mais son adresse dans le tas. Ainsi, on peut écrire des fonctions et procédures qui modifient des listes passées en paramètres.

```
def changeFst(l, a):
    l[0]=a
    return l

t = [1,2,3]
changeFst(t, 0)

```

À la fin de l'exécution du programme ci-dessus, la liste `t` contient la liste `[0,2,3]`. Ceci ne contredit pas la règle de localité. En effet, le paramètre `t` n'a pas été modifié : il pointe toujours vers la même adresse dans le tas. Par contre, ses éléments ont été modifiés.

Exercice 3 (Exécution, ⋆)
Renommez les variables et ajouter le mot-clé `global` pour lever les ambiguités.

```
a = 0
def f (a, b):
    c = 1
    a = a+b
    return a+c

def g (b):
    a = a+b
    b = 1
    c = a
    return f (a, b)
```

□

Exercice 4 (compte en banque, ⋆⋆)
On considère le programme suivant.

```
balance = 0

def deposit(amount):
    global balance
    balance = balance + amount

def withdraw(amount):
    global balance
    newbalance = balance - amount
    if (newbalance>0):
        balance = newbalance
    else:
        amount = balance
        balance = 0
    return amount

print(balance)
deposit(1000)
print(balance)
amount1 = 100
print(withdraw(amount1))
```

```
print(balance)
amount2 = 1000
print(withdraw(amount2))
deposit(1000)
amount = 2000
print(withdraw(amount))
print(amount)
```

code/exoCompte.py

1. *Quelles sont les variables globales, les paramètres, et les variables locales ?*
2. *Qu'affiche le programme ?*

□

Exercice 5 (test, ⋆⋆⋆)

1. *Écrire une fonction* `isInMatrix` *qui prend en argument une liste* `M` *de listes d'entiers et un entier* `n` *et qui renvoie* `True` *si* `n` *apparait dans* `M`.

   Contrat:
   Pour $0 \leq n \leq 5$ *et* $6 \leq m \leq 9$.
   `M1 = [[0,1,2],[3,4,5]]` *et* `M2 = [[0,5],[1,4],[2,2],[3, 1]]`.

   ```
   isInMatrix([[]], n)  →  False
     isInMatrix(M1, n)  →  True
     isInMatrix(M1, m)  →  False
    isInMatrix(M2 , n)  →  True
     isInMatrix(M2, m)  →  False
   ```

2. *Écrire les tests correspondants au contrat ci-dessus.*
3. *Afin de factoriser le code et de ne pas répéter* `x` *fois les tests, proposer un encodage pour les tests et une boucle permettant d'afficher le résultat des tests. Afficher les tests pour M1 et les entiers compris entre 0 et 9 inclus.*
4. *Afin de factoriser le code et de ne pas recoder systématiquement la boucle de test, on propose d'écrire une fonction.*
   Écrire une fonction test qui prend en argument une fonction et une liste de tests et qui renvoie une liste de résultats. Chaque test représente l'entrée sur laquelle tester la fonction et la sortie attendue. La fonction renverra une liste de résultats représentant l'entrée, la sortie attendue et la sortie obtenue.

   Contrat:
   Par exemple, on suppose donnée une fonction `f` *qui devrait renvoyer le double de son argument, mais qui n'a pas le comportement attendu...*

   ```
   test(f, [(0,0),(1, 2),(5,10)])  →  [(0,0,1), (1,2,2), (5,10,10)]
   ```

   Quels sont les paramètres et variables locales à la fonction `test` *?*
5. *Tester la fonction test sur la fonction* `isInMatrix` *et les paramètres définis dans les contrats.*
6. *Afin de factoriser l'affichage des tests, écrire une procédure* `Show` *qui affiche les entrées, les sorties attendues et les sorties obtenues pour chacun des échecs.*
7. *Ajouter deux variables globales* `n_test` *et* `n_failure` *qui permettent d'ajouter le nombre de tests effectués et le nombre de tests ratés.*
8. *Modifier la procédure* `show` *afin qu'elle affiche le nombre total de tests et de ratés depuis le début du programme.*

□

Documentation et Spécification [Cours]

Un programme peu ou mal documenté est difficile à corriger, modifier et réutiliser. Il est donc important de documenter et de commenter systématiquement son code.

Pour chaque **fonction**, avant l'en-tête, on place quelques lignes de commentaires, au format suivant :

- une ligne pour chaque paramètre d'entrée, indiquant son type, ce que ce paramètre représente et une propriété attendue par la fontion sur l'entrée, que l'on appelle **précondition** ;
- une ligne pour la valeur renvoyée, son type, ce qu'elle représente et la propriété promise par la fonction sur cette sortie, que l'on appelle **postcondition**.

Des commentaires supplémentaires liés à l'implémentation de la fonction seront placés dans son corps, comme dans l'exemple suivant :

```
# Verification de la correction d'une date.
#
# Entree : d un entier designant le jour.
# Entree : m un entier designant le mois.
# Entree : y un entier designant l'annee.
# Sortie : True si la date est correcte, False sinon.

def checkDate (d, m, y):
    if (m >= 1 and m <= 12 and d >= 1 and d <= 31):
        if (m == 4 or m == 6 or m == 9 or m == 11):
            return (d <= 30)
        else:
            if (m == 2): # Est-ce une annee bissextile?
                if ((y%4 == 0 and y%100 !=0 ) or y%400 == 0):
                    return (d <= 29)
                else:
                    return (d <= 28)
            else:
                return True
    else:
        return False
```

Pour les **procédures**, on indiquera **l'effet qui sera observé** (affichage, modification d'une liste, modification de variables globales, lecture d'un fichier,...) après l'exécution de la procédure à la place de la **postcondition**, qui n'a pas lieu d'être puisqu'aucune valeur n'est renvoyée par une procédure.

```
#
# Procedure LastFirst
#
# Entree: t une liste d'entiers
#
# Effet1: mise a 0 du dernier element de t
# Effet2: affichage du premier element de t
#

def LastFirst(t):
    t[-1] = 0
    print(t[0])
```

La spécification des fonctions et procédures ainsi fournie doit permettre de les utiliser sans avoir à connaître leur code.

Exercice 6 (Inclusion de listes, $\star$)

Écrire une fonction qui implémente la spécification suivante :

```
#
# Inclusion d'une liste dans un autre.
#
# Entree : une liste d'entiers a
# Entree : une liste d'entiers b
# Sortie : True si et seulement si tous les elements de la liste a
# sont contenus dans la liste b en prenant en compte
# les repetitions.
#
```

□

Exercice 7 (Séquences dans une liste, $\star$)
Écrire une fonction qui implémente la spécification suivante :

```
#
# Entrée : une liste d'entiers a.
# Sortie : True si et seulement si a contient deux
# sequences consécutives identiques. Par exemple, sur
# [1,2,3,2,3,7] la valeur à renvoyer est True.
#
```

□

2 DIY

Dans les exercices suivants, vous vous appliquerez à bien factoriser votre code.

Exercice 8 (Réaliser des spécifications, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
# Somme des entiers d'un intervalle.
#
# Entree: low un entier designant le debut de l'intervalle.
# Entree: high un entier designant la fin de l'intervalle.
# Sortie: la somme des entiers compris au sens large entre
# low et high.
```

□

Exercice 9 (Utiliser des spécifications, ⋆⋆)

La conjecture de Goldbach *dit que tout nombre pair plus grand que 2 peut être écrit comme la somme de deux nombres premiers.*

1. *Écrire une fonction* `boolean isPrime(int p)` *qui implémente la spécification suivante :*

```
# Teste si un entier est premier.
#
# Entree : un entier p plus grand que 1.
# Sortie : True si et seulement si n est premier, False sinon.
```

2. *Écrire la spécification de la fonction* `isGoldbach(n)` *qui renvoie* `True` *si et seulement si la conjecture est vérifiée pour l'entier* `n`. *(Si* `n` *n'est pas pair ou est strictement plus petit que 2 la fonction renvoie* `True`*.)*
3. *Donner le code de la fonction* `isGoldbach(n)`. *(On utilisera la fonction précédemment définie.)*
4. *Écrire une fonction* `goldbach(p)` *qui implémente la spécification suivante. (On utilisera la fonction précédemment définie.) :*

```
# Verifie la conjecture de Goldbach jusqu'a un certain rang.
#
# Entree : un entier p.
# Sortie : True si et seulement si tout entier inferieur ou
# egal a p verifie la conjecture de Goldbach.
#
```

□

Exercice 10 (Chaîne correspondante à une liste, ⋆)
Écrire une fonction qui implémente la spécification suivante :

```
# Entree : une liste de listes d'entiers t.
# Sortie : une chaine de caracteres le representant.
#
# Par exemple, si t vaut [[1,2],[3],[2,3,7]], la chaine de caracteres
# sera "[[1,2],[3],[2,3,7]]".
#
```

□

Exercice 11 (Tri d'une liste, ⋆⋆)

1. *Écrire une procédure* `swap` *qui implémente la spécification suivante :*

```
#
# Echange le contenu de deux cases d'une liste.
#
# Entree : une liste d'entiers t.
# Entree : un entier i compris dans les bornes de la liste.
# Entree : un entier j compris dans les bornes de la liste.
#
# Apres avoir execute cette procedure les contenus des
# cases i et j de la liste t sont echanges.
#
```

2. *En utilisant la procédure précédente, écrire une procédure* `findmin` *de spécification suivante :*

```
#
# Echange le contenu de la case i avec la case j de la liste t qui
# contient le plus petit entier des cases d'indice compris entre
# i et l'indice de fin de la liste.
#
# Entree : une liste d'entiers t.
# Entree : un entier i compris dans les bornes de la liste.
#
#
```

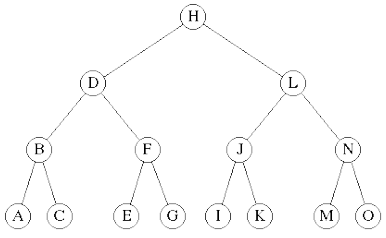
3. *En utilisant la procédure précédente, écrire une procédure* `sort` *qui implémente la spécification suivante :*

```
#
# Tri la liste.
#
# Entree : une liste d'entiers t.
#
```

□

Exercice 12 (Tournoi, ⋆⋆⋆)

Un tournoi est un arbre binaire (les nœuds de l'arbre contiennent toujours deux fils) dont chaque noeud interne correspond au gagnant du match entre ces deux fils. Par exemple, dans le tournoi



on a $B = \max(A, C)$, $D = \max(B, F)$, *etc. Les participants au tournoi sont les entiers apparaissant sur les feuilles de l'arbre ; dans l'exemple ci-dessus, les participants sont donc les entiers* A, C, E, G, I, K, M, O. *Pour simplifier le problème, le nombre de participants est supposé être une puissance de 2.*

Un arbre est représenté par une liste en décidant que :

— *la racine de l'arbre est à la position 0 ;*
— *le fils gauche d'un nœud situé dans la case* i *est situé dans la case* $2 * i + 1$ *;*

— *le fils droit d'un nœud situé dans la case i est situé dans la case $2 * i + 2$.*

Ainsi, l'arbre précédent peut être représenté par la liste de taille 15 suivant :

```
[ H, D, L, B, F, J, N, A, C, E, G, I, K, M, O ]
```

Pour représenter un tournoi avec 2^n participants, il faut $2^{n+1} - 1$ noeuds dans l'arbre. (Montrez-le par récurrence pour vous en convaincre.) Il faut donc créer une liste de taille $2^{n+1} - 1$ et on l'initialise avec la valeur -1 pour indiquer qu'aucun match n'a encore eu lieu.

Pour commencer le tournoi, on place les participants sur les feuilles de l'arbre. Tant qu'il ne reste pas qu'un seul participant, on fait un tour supplémentaire de jeu. Un tour de jeu consiste à faire jouer tous les matchs du niveau le plus bas de l'arbre dont les matchs ne sont pas encore joués. On fait alors "remonter" le gagnant de chaque match joué pour ce tour au niveau suivant, ce qui a pour effet de préparer le tour suivant, joué au niveau du dessus dans l'arbre.

1. *Après avoir spécifié et implémenté des fonctions utiles pour :*
   — *construire l'arbre nécessaire à la représentation d'un tournoi entre 2^n participants représentés par une liste ;*
   — *représenter l'indice du père d'un nœud dans l'arbre ;*
   — *faire jouer deux participants en faisant remonter le gagnant dans l'arbre ;*
   Écrivez une fonction `tournament` *de spécification suivante :*

```
#
#
# Entree : un entier n.
# Entree : une liste d'entiers t de taille 2 puissance n.
# Sortie : une liste representant le tournoi des participants
# de t.
#
#
```

2. *Comment généraliser la fonction précédente à un nombre de participants qui n'est pas une puissance de 2 ?*
3. *Après avoir spécifié et implémenté une fonction qui calcule l'ensemble des participants qui ont perdu contre le vainqueur d'un tournoi, écrivez une fonction* `kbest` *de spécification suivante :*

```
#
# Entree : une liste d'entiers t.
# Entree : un entier k compris entre 1 et la taille de la liste.
# Sortie : une liste representant les k meilleurs participants
# de t.
#
```

□

Introduction à la Programmation 1 PYTHON

MT11Y060

Séance 9 de cours/TD

Université de Paris

Objectifs:

— Comprendre ce qu'est la factorisation de code.

— Réflechir sur la qualité d'un code

1 Factoriser du code

Factorisation de code [COURS]

Dans la communication, en particulier écrite, la concision est une vertu qui va souvent de pair avec la clarté et l'élégance. Par exemple, on préfère écrire l'expression factorisée :

$7 \times (x + y + z) \times w$

plutôt que l'expression développée :

$w \times x \times 7 + w \times 7 \times y + 7 \times z \times w$.

La deuxième expression est équivalente à la première, mais plus difficile à comprendre.

Dans la pratique de la programmation, produire du code concis, clair, facilement réutilisable et documenté est très important. En effet, la programmation est certes un exercice de communication avec la machine, mais aussi un exercice de communication avec l'humain qui essaiera de comprendre votre programme (pour l'améliorer, pour corriger une erreur ou encore pour le réutiliser dans un autre contexte).

Il arrive souvent que l'on écrive un premier programme qui fonctionne mais n'est pas très bien écrit. On cherche alors à le *réécrire en un programme équivalent* mais plus concis. Cela s'appelle **factoriser** du code.

Il existe bien sûr une infinité de façons de factoriser du code. S'il n'y a pas de recette universelle pour écrire du code concis et élégant, il existe des transformations récurrentes comme par exemple :

— **Déplacer** des instructions pour éviter de les répéter plusieurs fois ;
— **Réécrire** des conditions pour regrouper des cas ;
— **Factoriser** plusieurs instructions répétées à l'aide d'une boucle ;
— **Découper** une longue série d'instructions en plusieurs fonctions ou procédures ;
— **Généraliser** plusieurs blocs d'instructions similaires en introduisant des fonctions ou des procédures.

Exemple : un code très verbeux [Cours]

Voici un programme verbeux formé d'une série d'instructions conditionnelles difficiles à comprendre :

```
if (dir == 1) :
    x = x - 1
    y = y + 1
    drawPosition (x, y)
elif (dir == 2) :
    y = y + 1
    drawPosition (x, y)
elif (dir == 3) :
    x = x + 1
    y = y + 1
    drawPosition (x, y)
elif (dir == 4) :
    x = x - 1
    drawPosition (x, y)
elif (dir == 5) :
    x = x + 1
    drawPosition (x, y)
elif (dir == 6) :
    x = x - 1
    y = y - 1
    drawPosition (x, y)
elif (dir == 7) :
    y = y - 1
    drawPosition (x, y)
elif (dir == 8) :
    x = x + 1
    y = y - 1
    drawPosition (x, y)
```

Listing 1 – Exemple de code verbeux

Dans cet exemple, `dir` est une variable de type `int` qui représente une direction dans le plan :

1	2	3
4	X	5
6	7	8

L'objectif de ce fragment de programme est d'afficher un point à la position `(x, y)` une fois qu'il a été déplacé dans la direction `dir`.

Ce programme a plusieurs défauts :

1. C'est une longue série de tests : comment se convaincre qu'aucun test n'a été oublié ?
2. Les instructions de chaque branche se ressemblent sans être exactement les mêmes : comment se convaincre que ces différences sont bien justifiées ?
3. Le programme fait trois choses en même temps : (i) il calcule un déplacement en `x` et `y` en fonction de la valeur de `dir` ; (ii) il met à jour la valeur de `x` et `y` ; (iii) il affiche le point à la nouvelle position. Pour être plus compréhensible, ce programme devrait expliciter ces trois étapes.

Déplacement d'instructions [Cours]

Dans le programme précédent, l'appel de procédure `drawPosition (x, y)` apparaît huit fois en dernière position de la séquence d'instructions de chaque branche. On peut sans risque *déplacer cette instruction après la série des tests pour n'avoir à l'écrire qu'une fois.*

```
if (dir == 1) :
    x = x - 1
    y = y + 1
elif (dir == 2) :
    y = y + 1
elif (dir == 3) :
    x = x + 1
    y = y + 1
elif (dir == 4) :
    x = x - 1
elif (dir == 5) :
    x = x + 1
elif (dir == 6) :
    x = x - 1
    y = y - 1
elif (dir == 7) :
    y = y - 1
elif (dir == 8) :
    x = x + 1
    y = y - 1
drawPosition (x, y)
```

Listing 2 – Après déplacement d'une instruction.

Explicitation de variables [Cours]

On peut faire apparaître les trois étapes du programme en introduisant des variables `dx` et `dy` qui représentent le déplacement de `x` et `y` en fonction de la valeur de `dir`.

```
dx = 0
dy = 0
if (dir == 1) :
    dx = -1
    dy = 1
elif (dir == 2) :
    dy = 1
elif (dir == 3) :
    dx = 1
    dy = 1
elif (dir == 4) :
    dx = -1
elif (dir == 5) :
    dx = 1
elif (dir == 6) :
    dx = -1
    dy = -1
elif (dir == 7) :
    dy = -1
elif (dir == 8) :
    dx = 1
    dy = -1
```

```
x = x + dx
y = y + dy
drawPosition (x, y)
```

Listing 3 – Après introduction de deux variables.

Regroupement de tests [Cours]

Dans beaucoup de cas, les calculs de `dx` et de `dy` sont les mêmes. On peut restructurer les tests comme suit :

```
dx = 0
dy = 0
if (dir == 1 or dir == 2 or dir == 3) :
    dy = 1
elif (dir == 6 or dir == 7 or dir == 8) :
    dy = -1
if (dir == 3 or dir == 5 or dir == 8) :
    dx = 1
elif (dir == 1 or dir == 4 or dir == 6) :
    dx = -1
x = x + dx
y = y + dy
drawPosition (x, y)
```

Listing 4 – Après regroupement des tests.

Introduction de fonctions [Cours]

Le programme précédent est concis mais on peut encore améliorer sa lisibilité en introduisant du *vocabulaire* permettant de le décrire en des termes plus abstraits qu'avec les instructions basiques de Python. Pour introduire du vocabulaire, on peut se donner des fonctions aux noms bien choisis.

```
def dyFromDirection (dir) :
    if (dir == 1 or dir == 2 or dir == 3) :
        return 1
    if (dir == 6 or dir == 7 or dir == 8) :
        return -1
    return 0

def dxFromDirection (dir) :
    if (dir == 3 or dir == 5 or dir == 8) :
        return 1
    if (dir == 1 or dir == 4 or dir == 6) :
        return -1
    return 0
```

Listing 5 – Des fonctions, du nouveau vocabulaire.

Dès lors le code final se réduit à trois lignes, aisément compréhensibles :

```
x = x + dxFromDirection (dir)
y = y + dyFromDirection (dir)
drawPosition (x, y)
```

Listing 6 – Le programme final.

Exercice 1 (Bien choisir les tests, ⋆⋆)
Réecrire le fragment de code suivant, de manière concise :

```
if (x < 0) :
```

```
    y = y + 1
    z = z - 1
else :
    if (1 <= x and x < 5) :
        y = y - 1
        z = z - 1
    else :
        if (6 <= x and x < 10) :
            y = y + 1
            z = z - 1
        else :
            y = y - 1
            z = z - 1
```

□

Factorisation à l'aide d'une boucle [Cours]

Il est toujours préférable de remplacer par une boucle une série d'instructions répétées un grand nombre de fois. Cela peut augmenter le nombre de lignes de code mais la boucle obtenue est plus compréhensible. Par ailleurs, lorsque le programmeur doit taper une longue série d'instructions très similaires, il est plus probable qu'il fasse une erreur que lorsqu'il écrit une seule boucle.

Ainsi, prendre la moyenne d'une liste d'entiers `array` peut s'écrire ainsi :

```
mean = (array[0] + array[1] + array[2] + array[3] + array[4]
        + array[5] + array[6] + array[7] + array[8] + array[9]) // 10
```

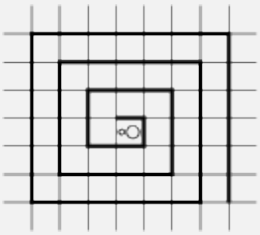
Listing 7 – Calcul de moyenne verbeux

mais on préférera l'usage d'une boucle :

```
mean = 0
for i in range(0, 10, 1) :
    mean = mean + array[i]
mean = mean // 10
```

Listing 8 – Calcul de moyenne concis

Parfois le caractère répétitif de la tâche à accomplir est évident, mais bien paramétrer les boucles censées factoriser le code ne l'est pas. Considérez le labyrinthe :



```
west()
north()
east()
east()
south()
south()
west()
west()
west()
north()
north()
north()
east()
east()
east()
east()
south()
south()
south()
south()
west()
west()
west()
west()
west()
north()
north()
north()
north()
north()
east()
east()
east()
east()
east()
east()
south()
south()
south()
south()
south()
south()
```

Listing 9 – Sortie compliquée du labyrinthe.

En utilisant des boucles :

```
for i in range(1, 4, 1) :
    for j in range(1, 2 * i, 1) :
        west()
    for j in range(1, 2 * i, 1) :
        north()
    for j in range(1, 2 * i + 1, 1) :
        east()
    for j in range(1, 2 * i + 1, 1) :
        south()
```

Listing 10 – Sortie claire du labyrinthe.

[Cours]

Souvent, quand on passe du code verbeux au code concis, on cerne mieux la nature du problème à résoudre : le code verbeux se focalise sur *une instance* du problème, le code concis sur son *essence*. À partir de là, on peut définir un outil qui résout toutes les instances du problème en question, c'est à dire une fonction (ou une procédure) dont les paramètres décrivent l'instance du problème et qui renvoie le résultat escompté (ou exécute la tâche escomptée). Dans l'exemple de la moyenne de la liste d'entiers, cela donne :

```
def average (a) :
    acc = 0
    l = len(a)
    for i in range(0, l, 1) :
        acc = acc + a[i]
    return (acc // l)
```

Listing 11 – Calcul de la moyenne par une fonction

Ensuite, quand on veut affecter à la variable entiere `mean` la (partie entière de la) moyenne de la liste d'entiers `array`, on appelle la fonction comme suit :

```
mean = average (array)
```

Dans le cas du labyrinthe :

```
def exitLabySpiral (size) :
    for i in range(1, size // 2 + 1, 1) :
        for j in range(1, 2 * i, 1) :
            west ()
        for j in range(1, 2 * i, 1) :
            north ()
        for j in range(1, 2 * i + 1, 1) ;
            east ()
        for j in range(1, 2 * i + 1, 1) :
            south ()
```

Listing 12 – Sortie du labyrinthe par une procédure

Ensuite, on peut utiliser cette procédure pour sortir d'un labyrinthe à spirale de `n` lignes, en l'appelant comme suit :

```
exitLabySpiral (n)
```

Dans le cas de la sortie du labyrinthe à spirale, il est possible de factoriser ultérieurement le code en définissant une procédure qui permet à la fourmi d'avancer d'un nombre de cases donné dans une direction donnée. Le type du premier paramètre, le nombre de cases, est `int`. Pour le deuxième, la direction, on pourrait aussi choisir `int`, à condition de choisir un code entier pour chaque point cardinal. Ce deuxième choix donne le code suivant :

```
1
2  # Procedure qui fait avancer la fourmi d'un nombre de cases p
3  # dans la direction d. La direction est codÃ©e comme suit
4  # nord = 1, est = 2, sud = 3, ouest = 4
5
6  def goForward (p,d) :
7      for i in range(1, p+1, 1) :
8          if (d == 1) :
9              north ()
10         elif (d == 2) :
11             east ()
12         elif (d == 3) :
13             south ()
14         elif (d == 4) :
15             west ()
```

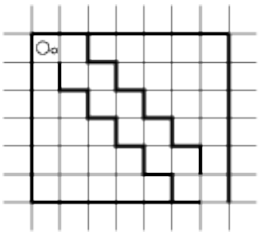
On peut à présent utiliser `goForward` pour écrire une deuxième version de `exitLabySpyral` :

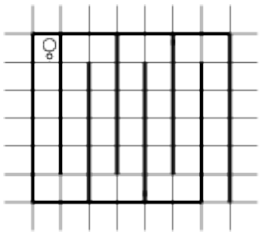
```
1  def exitLabySpiral2(size) :
2      for i in range(1, size // 2 + 1, 1) :
3          goForward (2 * i - 1, 4)
4          goForward (2 * i - 1, 1)
5          goForward (2 * i, 2)
6          goForward (2 * i, 3)
```

Exercice 2 (Labyrinthes, ⋆⋆)

On considère le problème de sortir des labyrinthes ci-dessous. (On dispose des procédures `north()`, `south()`, `east()`, `west()` *pour déplacer la fourmi.) Donnez pour chacun :*

1. *le code verbeux,*
2. *le code concis en utilisant une boucle,*
3. *une fonction qui prend en paramètre la taille du labyrinthe et résout le problème.*





□

Introduction à la Programmation 1 PYTHON

MT11Y060

Séance 10 de cours/TD

Université de Paris

Objectifs:

- Les tableaux associatifs
- Les générateurs

[COURS]

Dans ce cours nous présentons quelques sujets un peu avancés de la programmation en Python.

Ce cours est optionnel et ne fait pas partie des sujets de l'examen.

1 Les tableaux associatifs

Pourquoi des tableaux associatifs [COURS]

- Nous avons dans ce semestre beaucoup travaillé avec les listes. On peut voir une liste d'entiers l comme une fonction qui a le domaine $\{n \mid 0 \leq n < len(l)\}$, et le codomaine $\mathbb{N}$. La liste `[73,17,42]`, par exemple, peut être vue comme une fonction avec le domaine $\{0, 1, 2\}$, et qui associe 73 à 0, 17 à 1, et 42 à 2. Il s'agit même des fonctions *modifiables*, car on peut *modifier* par une affectation la valeur contenue dans une case d'une liste.
- Puis nous avons vu que les listes sont en fait plus générales : on n'a pas seulement des fonctions vers les entiers, mais des fonctions vers n'importe quel autre type de valeurs. Par exemple, nous avons vu des listes de chaînes de caractères, des listes de listes d'entiers, etc.
- Les listes présentent quand même une restriction : le *domaine*, c'est-à-dire les indices d'une liste, sont toujours des entiers. Or, on souhaite parfois réaliser des fonctions avec un domaine différent, par exemple des chaînes de caractères : pensez par exemple à un annuaire qui associe au nom d'une personne un numéro de téléphone.
- Les *tableaux associatifs*, ou *dictionnaires* en anglais, permettent de réaliser des fonctions partielles dont le domaine est un ensemble fini de chaînes de caractères.
- Nous disons qu'un tableau associatif associe des *valeurs* à des *clefs*. Autrement dit, les clefs d'un tableau associatif d sont les éléments du domaine de d, vue comme une fonction partielle.

Comment travailler avec les tableaux associatifs [COURS]

- On peut créer un tableau associatif en listant tous les clefs, avec leurs valeurs associées. Par exemple

```
1 name = {"Lennart": "Hofstaedter", "Sheldon": "Cooper", "Howard":
    "Wolowitz"}
```

  défini un tableau associatif qui associe aux trois prénoms *Lennart*, *Sheldon*, et *Howard* des noms de famille.

— En particulier, le tableau vide (qui n'a aucune clef) est noté `{}`.
— On peut obtenir la valeur associée par un tableaux associatif d à une clef c en écrivant $d[c]$. Par exemple, l'expression `nom["Lennart"]` donne comme résultat `"Hofstaedter"`.
— Dans le cas des listes il était facile de savoir si un entier i est un indice légal d'une liste l : il suffisait de tester si $0 \leq i < len(l)$. Cela n'est pas possible pour un tableau associatif dont les clefs peuvent former un ensemble quelconque. Pour cette raison, Python propose une expression particulière pour tester si une chaîne s est une clef d'un tableaux associatif t : `s in t`. Par exemple,

```
if ("Sheldon" in nom):
  print("on a Sheldon")
else:
  print("pas de Sheldon !")

```

affiche `"on a Sheldon"`.
— `print` sait aussi afficher un tableau associatif. Par contre, l'ordre dans lequel un tableau est affiché n'est pas forcement l'ordre dans lequel on l'avait écrit. La raison pour cela est que l'ordre est simplement une question de représentation qui ne doit pas importer, et que Python choisit internement une représentation efficace qui peut utiliser un ordre différent.

Exercice 1 (Chercher dans des tableaux associatifs, $\star$)
Étant donnés un premier tableau associatif qui associe des noms à des prénoms de caractères, et un deuxième tableau associatif qui associe des noms d'acteurs à des noms de caractères :

```
nom = {"Lennart": "Hofstaedter", "Sheldon": "Cooper", "Howard": "
    Wolowitz"}
actor={"Cooper": "Parsons", "Hofstaedter": "Galecki"}
```

code/exoBigBang.py

1. *Écrire une fonction* `actor_of_character` *qui prend un prénom p en paramètre, et qui envoie le nom de l'acteur qui joue ce caractère. Par exemple,* `actor_of_character("Lennart")` *doit envoyer le résultat* `"Galecki"`.
2. *Améliorer votre fonction pour qu'elle envoie la chaîne* `"name unknown"` *dans le cas où p n'est pas un prénom connu d'un caractère, et envoie* `"actor unknown"` *dans le cas où le caractère est connu mais l'acteur qui le joue ne l'est pas. Par exemple,* `actor_of_character("Penny")` *doit envoyer* `"name unknown"`, *et* `actor_of_character("Howard")` *doit envoyer* `"actor unknown"`.

□

Exercice 2 (Un traducteur, $\star$)
Dans cet exercice nous allons écrire un traducteur français vers anglais très primitif. Étant donné un dictionnaire français-anglais, comme celui-ci :

```
dict={"langage": "language", "un": "a", "est": "is", "formidable": "
    wonderful"}
```

code/exoTranslator.py

Une phrase va être représentée par une liste de mots. Écrire une fonction `translate` *qui prend une phrase française en paramètre, et qui envoie une nouvelle phrase en anglais obtenue par une traduction mot par mot. Quand un mot français n'est pas trouvé dans le dictionnaire, on garde simplement le mot français. Ainsi,* `(translate(["Python", "est", "formidable"])` *doit donner le résultat* `["Python", "is", "wonderful"]`.

□

Plus sur les tableaux associatifs [Cours]

— On peut modifier une valeur dans un tableau associatif de la même façon qu'on modifie le contenu d'une case dans une liste. Par exemple,

```
chef = {"Stark": "Ned", "Lannister": "Tywin"}
chef["Stark"] = "Robert"
print(chef)

```

affiche

```
{"Stark": "Robert", "Lannister": "Tywin"}

```

— Une différence essentielle entre les tableaux associatifs et les listes vues ici en cours est le fait que les tableaux associatifs sont *extensibles*. Ça veut dire qu'on peut aussi faire une affectation dans un tableau associatif pour une clef qui n'existe pas encore : la nouvelle clef est alors ajoutée au tableau. Par exemple,

```
chef = {"Stark": "Ned", "Lannister": "Tywin"}
chef["Frey"] = "Walder"
print(chef)

```

affiche

```
{"Stark": "Ned", "Lannister": "Tywin", "Frey": "Walder"}

```

— En vérité, les tableaux associatifs peuvent aussi avoir des clefs qui sont d'autres types que des chaînes de caractère, mais c'est rarement utilisé, et nous n'avons pas encore vu les autres types qui peuvent être utilisés comme type de clefs.

— Dans d'autres langages de programmation, les tableaux associatifs sont souvent appelés des *tables de hachage*.

2 Les générateurs

Différents types de générateurs [Cours]

— Nous avons vu la construction `range(start,end,step)` dans le cadre des boucles `for`. Il est temps de se demander quelle est la nature de l'objet construit par cette expression. Il s'agit d'un *générateur* : c'est un objet abstrait auquel on peut demander successivement de générer une nouvelle valeur. Il n'est pas nécessaire de savoir comment faire ces demandes à un générateur ; nous allons utiliser les générateurs seulement dans le contexte des boucles `for`, et ces boucles savent interagir avec un générateur pour obtenir de lui toutes les valeurs au fur et à mesure.
Il y a en Python des générateurs autres que `range` qui sont utiles à connaître, et qu'on peut utiliser à la place de `range` dans une boucle `for`.

— On peut utiliser une liste comme un générateur : les valeurs générées sont les éléments de la liste. Par exemple

```
for n in [2,3,4] :
  print(n*n)

```

affiche
4

9
16

— On peut utiliser une chaîne de caractères comme un générateur : les valeurs sont les caractères qui constituent la chaînes. Par exemple

```
s=""
for c in "miroir":
  s=c+s
print(s)
```

code/exampleStrings.py

affiche la chaîne "riorim"

— On peut lire toutes les lignes d'un fichier à l'aide d'un générateur `open` qui prend le nom du fichier en argument. Par exemple,

```
for l in open("exampleFiles.py"):
  print(l,end="")
```

code/exampleFiles.py

affiche simplement le contenu du fichier `exo1.py`, Attention, chaque ligne envoyée par le générateur contient aussi le saut de ligne final (quand il y en a un dans le fichier), pour cette raison l'exemple affiche la ligne avec un `end=""`.

— On peut aussi utiliser un tableau associatif comme générateur : les valeurs envoyées sont les clefs utilisées dans le tableau, mais dans un ordre aléatoire. Par exemple

```
for c in {"hello" : "bonjour", "good bye" : "au revoir"}:
      print(c)
```

code/exampleKeys.py

affiche les deux lignes "hello" et "good bye", dans un ordre aléatoire.

Exercice 3 (Utiliser les générateurs, ⋆)

1. *Écrire une fonction qui prend une chaîne de caractères en paramètre, et qui envoie une copie de la même chaîne mais sans les sauts de ligne (caractère `"\n"` en python).*
2. *Écrire une fonction* `longueurs` *qui prend le nom d'un fichier en paramètre, et qui lit le fichier ligne par ligne. La fonction envoie un tableau associatif qui à chaque ligne trouvée dans le fichier associe sa longueur. Attention aux sauts de lignes qui ne doivent pas compter pour la longueur. Par exemple, si le fichier `montexte` a le contenu suivant :*

```
Longtemps,
je me suis
couché
de bonne heure.
```

code/montexte

alors l'appel `longueurs("montexte")` *doit envoyer le tableau associatif suivant :*

```
{"je me suis": 10, "Longtemps,": 10, "de bonne heure.": 15, "couché
    ": 6}

```

□

Exercice 4 (Travailler avec des dictionnaires, ⋆⋆)

1. *Écrire une fonction* `reverse` *qui prend un dictionnaire en paramètre, et qui renvoie un dictionnaire dans le sens inverse. Par exemple, appliquée au dictionnaire français-anglais suivant :*

```
{"maison": "house", "voiture": "car", "rue": "road", "car": "coach"
    }
```

```
2
```

la fonction doit renvoyer le dictionnaire anglais-français suivant :

```
1 {"car": "voiture", "road": "rue", "coach": "car", "house": "maison"
      }
2
```

2. *Écrire une procédure* `fauxAmis` *qui prend un dictionnaire en paramètre, et qui affiche tous les faux amis qui existent dans ce dictionnaire. Un faux ami dans un dictionnaire d'une langue source vers une langue cible est un mot du langage cible qui existe aussi dans la langue source, mais avec un sens différent. Par exemple, appliquée au dictionnaire français-anglais de la question précédente, la procédure doit afficher :*
   `Faux ami : car`
3. *Écrire une fonction qui prend deux dictionnaires en paramètre, et qui renvoie le dictionnaire obtenu en appliquant d'abord la traduction selon le premier dictionnaire, puis sur le mot obtenu par la traduction selon le deuxième dictionnaire quand elle est possible. Par exemple, appliquée au dictionnaire français-anglais de la première question, et le dictionnaire anglais-allemand suivant :*

```
1 dictED={"house": "Haus", "road": "Strasse", "tower": "Turm"}
2
```

la fonction doit envoyer le dictionnaire français-allemand suivant :

```
1 {"maison": "Haus", "rue": "Strasse"}
2
```

□