

Cours 3 : Tableaux et structures de contrôle



Ing-info — Ingénierie informatique

Christophe Gonzales

Plan du cours n° 3

- 1 Les tableaux
- 2 Approfondissements sur les variables
- 3 Les alternatives
- 4 Les boucles

Cours 3 : Tableaux et structures de contrôle

2/36

Les tableaux

- ▶ Vecteur en maths = tableau à 1 dimension en info

maths : (3,7,4) = info :

| | | |
|---|---|---|
| 3 | 7 | 4 |
|---|---|---|

↑
— valeur d'indice 0

Tableau en informatique

- ▶ Ensemble de valeurs de **même type** placées les unes à la suite des autres en mémoire.
- ▶ Accès aux valeurs via l'opérateur []
- ▶ 1er élément : tableau[0] — ⚠ indice = 0!
- ▶ 2ème élément : tableau[1], etc.
- ▶ Python ne connaît pas la notion de tableau 2D!
- ▶ Mais le package **numpy** définit les tableaux 2D 🤔

Plus fort : les tableaux à 2 dimensions !

- ▶ Matrice en maths = tableau à 2 dimensions en info

maths : $\begin{pmatrix} 2 & 3 & 5 \\ 4 & 1 & 8 \end{pmatrix}$ = info :

| | | |
|---|---|---|
| 2 | 3 | 5 |
| 4 | 1 | 8 |

↑
— matrice[1,0]

- ▶ Accès aux valeurs : matrice[i,j]
 - ▶ i = indice de ligne (commence à 0)
 - ▶ j = indice de colonne (commence à 0)
- ▶ En mémoire, toutes les lignes se suivent

Cours 3 : Tableaux et structures de contrôle

3/36

Cours 3 : Tableaux et structures de contrôle

4/36

Manipulation de tableaux

- Utiliser les tableaux \implies charger le package numpy :

```
import numpy as np
```

- Créer un tableau : plein de manières...
- Créer un tableau de 25 entiers égaux à 0 :

```
mon_tableau = np.full(25, 0)
```

- Affecter la valeur 1 à la cellule d'indice 3 :

```
mon_tableau[3] = 1
```

- Afficher la valeur de la cellule d'indice 3 :

```
print(mon_tableau[3])
```

- Afficher le contenu de tout le tableau :

```
print(mon_tableau)
```

Manipulation de tableaux à 2 dimensions

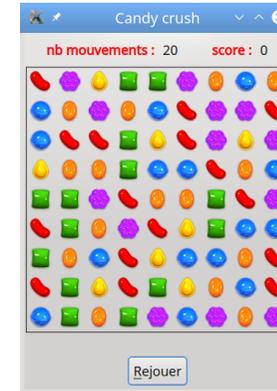
- Créer un tableau de 12 lignes \times 7 colonnes d'entiers égaux à 0 :

```
mon_tableau = np.full((12,7), 0)
```



bien mettre les dimensions 12,7 entre parenthèses !

- À quoi vont nous servir les tableaux ?



Exemple : transposée de matrice 2x2

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \implies A^T = \begin{pmatrix} a_{00} & a_{10} \\ a_{01} & a_{11} \end{pmatrix}$$

```
def transpose(A):  
    A_T = np.full((2,2), 0)
```

```
    A_T[0,0] = A[0,0]  
    A_T[0,1] = A[1,0]  
    A_T[1,0] = A[0,1]  
    A_T[1,1] = A[1,1]
```

```
    return A_T
```

```
A = np.array([[1,2],[3,4]])  
print(A)  
print()  
print(transpose(A))
```

```
[[1 2]  
 [3 4]]
```

```
[[1 3]  
 [2 4]]
```

Exemple : produit matrice-vecteur 2x2

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \implies \begin{cases} c_0 = a_{00} \times b_0 + a_{01} \times b_1 \\ c_1 = a_{10} \times b_0 + a_{11} \times b_1 \end{cases}$$

```
def prod_mat_vect(A,b):  
    c = np.full(2, 0)
```

```
    c[0] = A[0,0] * b[0] + A[0,1] * b[1]  
    c[1] = A[1,0] * b[0] + A[1,1] * b[1]
```

```
    return c
```

```
A = np.array([[1,2],[3,4]])  
b = np.array([2,3])
```

```
print(A, "\n")  
print(b, "\n")
```

```
print(prod_mat_vect(A,b))
```

```
[[1 2]  
 [3 4]]
```

```
[2 3]
```

```
[ 8 18]
```

Exemple : matrice 2x2 inversible ?

$$\det \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} = a_{00} \times a_{11} - a_{01} \times a_{10}$$

A inversible $\iff \det A \neq 0$

```
def det(A):
    return A[0,0] * A[1,1] - \
           A[0,1] * A[1,0]

def is_invertible(A):
    return det(A) != 0

A1 = np.array([[1,2],[3,4]])
print('A1:', is_invertible(A1))

A2 = np.array([[1,2],[2,4]])
print('A2:', is_invertible(A2))
```

A1: True
A2: False

Passage de tableaux en paramètres

► Tableau numpy : conteneur de valeurs

conteneurs mutables / non mutables

► mutable : après création, on peut modifier le **contenu**

► non mutable : après création, pas de modification possible

► **Exemples** : Chaînes de caractères : non mutables

Tableaux numpy : mutables

```
1 def modif_bof(tab):
2     tab = np.full(3, 0)
3
4     tab = np.full(2, 11)
5     modif_bof(tab)
6     print("bof :", tab)
7
8     def modif_ok(tab):
9         tab[1] = 0
10        modif_ok(tab)
11        print("ok :", tab)
```

bof : [11 11]
ok : [11 0]

► Dans une fonction : impossible de modifier la valeur d'un argument

► Mais possibilité de modifier les valeurs **contenues** dans des conteneurs **mutables**

\implies on peut modifier les valeurs des cellules des tableaux numpy

\implies animations de candy crush !

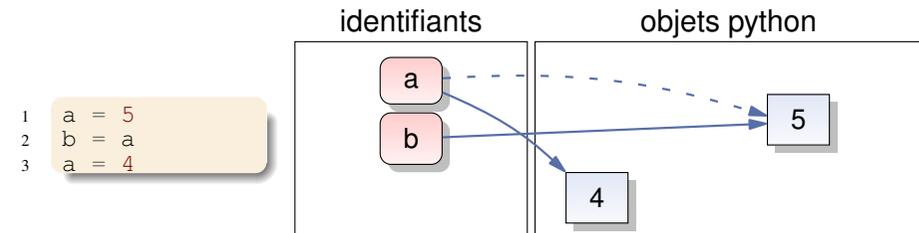
2 Approfondissements sur les variables

La « vraie » notion de variable en Python (1/3)

► Variable \implies nom + valeur (+ type)

► **Valeur** = « objet » python

► **Nom** = identifiant pointant vers l'objet

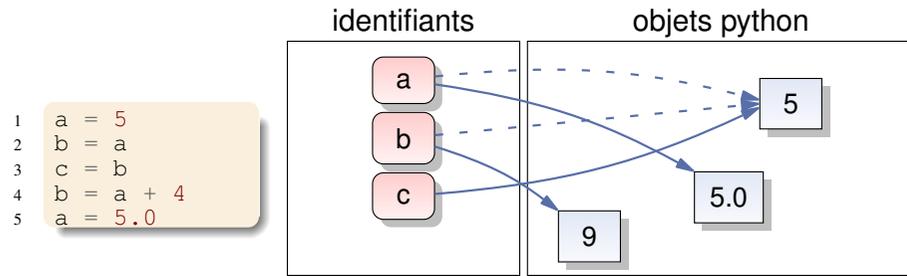


► **Règles d'affectation en Python :**

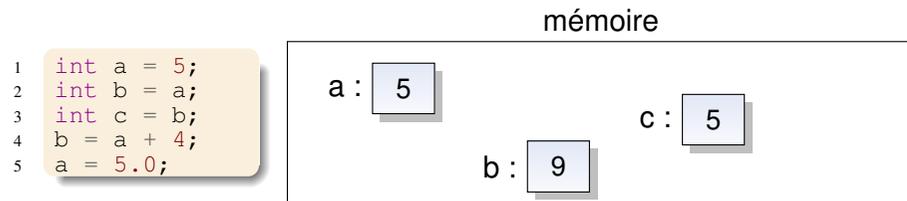
1 si la valeur n'est pas celle d'une autre variable alors :
on crée un nouvel objet avec cette valeur
sinon : on utilise l'objet de l'autre variable

2 on fait pointer le nom de variable vers l'objet

La « vraie » notion de variable en Python (2/3)



⚠ Dans d'autres langages, les règles sont différentes :
En C : variable \implies espace mémoire qui ne changera pas



La « vraie » notion de variable en Python (3/3)

► **Fonction id()** : entité d'un objet \approx emplacement mémoire

```

1 a = 500
2 print ("2 : id(a) =", id(a))
3
4 b = 500
5 print ("5 : id(b) =", id(b))
6
7 b = a
8 print ("8 : id(b) =", id(b))
9
10 a = a+1
11 print ("11: id(a) =", id(a))
12 print ("12: id(b) =", id(b))
        
```

```

2 : id(a) = 140527242493392
5 : id(b) = 140527242493904
8 : id(b) = 140527242493392
11: id(a) = 140527242493616
12: id(b) = 140527242493392
        
```

Mutable vs non mutable

► **Rappel du cours n° 2 :**

- **Objet mutable :** après création, on peut modifier sa valeur
tableaux numpy, listes, *etc.*
- **Objet non mutable :** après création, modification impossible
entiers, flottants, chaînes de caractères, booléens, *etc.*

```

1 import numpy as np
2
3 a = np.full (3,0)
4 print ("3 : id(a) =", id(a))
5
6 b = a
7 print ("7 : id(b) =", id(b))
8
9 a[1] = 4
10 print (a)
11 print ("11: id(a) =", id(a))
12
13 print (b)
14
15 a = np.full(3,1)
16 print ("16: id(a) =", id(a))
        
```

```

3 : id(a) = 140527268693696
7 : id(b) = 140527268693696
[0 4 0]
11: id(a) = 140527268693696
[0 4 0]
16: id(a) = 140527242320832
        
```

► **opérateur [] :** modifie la valeur de l'objet mutable

Passage de paramètres dans les fonctions (1/2)

► **Rappel du cours n° 2 : appel de fonction**

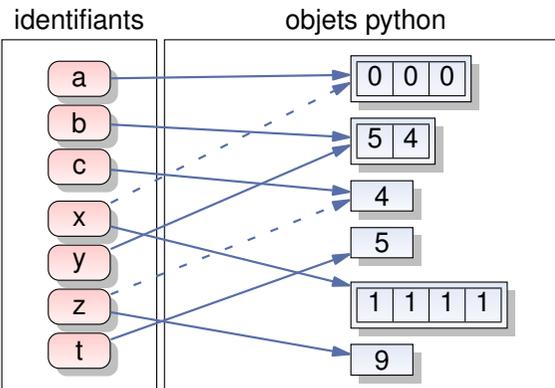
- 1 Évaluation des valeurs passées en argument de la fonction
- 2 Affectation de ces valeurs aux paramètres de la fonction
- 3 Exécution du code de la fonction
- 4 Récupération de la valeur retournée

► **Conséquences du 2 :**

- Valeur passée = objet \implies copie de pointeur
- Valeur passée \neq objet \implies création d'un nouvel objet

Passage de paramètres dans les fonctions (2/2)

```
1 import numpy as np
2
3 def f(x,y,z,t) :
4     x = np.full (4,1)
5     y[1] = 4
6     z += t
7
8 a = np.full(3,0)
9 b = np.full (2,5)
10 c = 4
11
12 f(a, b, c, c+1)
13 print (a,b,c)
```



```
[0 0 0] [5 4] 4
```

Modifications du contenu d'un tableau

```
1 def modif_bof(tab) :
2     tab = np.full(3,0)
3
4 tab = np.full(2,11)
5 modif_bof(tab)
6 print("bof :", tab)
7
8 def modif_ok(tab) :
9     tab[1] = 0
10 modif_ok(tab)
11 print("ok  :", tab)
```

```
bof : [11 11]
ok  : [11 0]
```

⇒ modifier les éléments un par un !!!

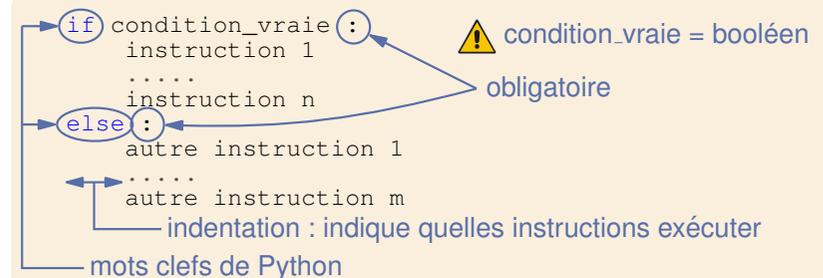
3 Alternatives

if/then/else

Problème : Déplacement de bonbon : vers la droite ou la gauche ?

```
Si ancien_x_bonbon < nouvel_x_bonbon alors
direction droite
Sinon
direction gauche
```

Dans les 2 cas, afficher la direction



► else : optionnel (à utiliser seulement si besoin)

if/then/else : exemples

► Déplacement bonbon :

```
if ancien_x_bonbon < nouvel_x_bonbon :
    direction = "droite"
else:
    direction = "gauche"

print(direction)
```

► Fonction qui renvoie la valeur absolue de son paramètre :

```
def val_absolue(x) :
    if x >= 0:
        return x
    else:
        return -x
```

► Fonction qui affiche 1 si le nombre saisi par un utilisateur est pair :

```
def saisie_pair_impair() :
    x = input("saisissez un nombre entier : ")
    if x % 2 == 0:
        print(1)
```

if/then/else : exemples (bis)

► Fonction qui renvoie le max de 2 éléments :

```
def max(x,y) :
    if x >= y:
        return x
    else:
        return y
```

► Fonction qui renvoie le max de 3 éléments :

```
def max3(x,y,z) :
    if x >= y:
        if x >= z:
            return x
        else:
            return z
    else:
        if y >= z:
            return y
        else:
            return z
```

```
def max3(x,y,z) :
    if x >= y and x >= z:
        return x
    else:
        if y >= x and y >= z:
            return y
        else:
            return z
```

elif : le limiteur d'indentation

elif

► elif \iff else if

```
def max3(x,y,z) :
    if x >= y and x >= z:
        return x
    else:
        if y >= x and y >= z:
            return y
        else:
            return z
```



```
def max3(x,y,z) :
    if x >= y and x >= z:
        return x
    elif y >= x and y >= z:
        return y
    else:
        return z
```

► On peut faire autant de elif que l'on veut !

```
if x > 1000:
    print("très grand")
else:
    if x > 100:
        print("grand")
    else:
        if x > 10:
            print("moyen")
        else:
            print("petit")
```



```
if x > 1000:
    print("très grand")
elif x > 100:
    print("grand")
elif x > 10:
    print("moyen")
else:
    print("petit")
```

4 Les boucles

Des boucles ? Pour quoi faire ?

Les boucles

- ▶ Servent à répéter des séquences d'instructions
- ▶ Répétition = **itération**
- ▶ Arrêt de la boucle : un booléen
⇒ au moment où on écrit le code, on ne connaît pas forcément le nombre d'itérations

▶ 2 types de boucles en Python :

- ▶ les boucles **while** (en français : tant que)
- ▶ les boucles **for**

 En Ing-info : pas de boucle for !

La boucle while

Syntaxe :

```
while condition_vraie : ← obligatoire
    instruction 1
    .
    .
    .
    instruction n
    ← indentation : indique quelles instructions exécuter
← mot clef de Python
```

▶ Exemple : afficher tous les entiers de 1 à 100

```
i = 1
while i <= 100 : # tant que i est inférieur à 100
    print(i)
    i = i + 1 ←  Ne pas oublier l'incrément !
```

- ▶ **Conseil** : Quand on écrit le while :
écrire tout de suite l'incrément
puis, entre les 2, les instructions de la boucle

Quelques exemples simples (1/2)

▶ Problème :

- ▶ Demander à l'utilisateur de saisir des nombres ≥ 0
- ▶ Continuer jusqu'à saisie d'un nombre < 0
- ▶ Afficher la somme des nombres positifs saisis

```
somme = 0
fin_saisie = False

while not fin_saisie :
    nombre = input("saisissez un nombre positif ou nul")
    if nombre >= 0:
        somme += nombre
    else:
        fin_saisie = True

print("somme =", somme)
```

Quelques exemples simples (2/2)

▶ Problème :

- ▶ Demander à l'utilisateur de saisir des nombres ≥ 0
- ▶ Continuer jusqu'à saisie d'un nombre < 0
- ▶ Afficher la **moyenne** des nombres ≥ 0 saisis

```
somme = 0
nb_nombres_saisis = 0
fin_saisie = False

while not fin_saisie :
    nombre = input("saisissez un nombre positif ou nul")
    if nombre >= 0:
        somme += nombre
        nb_nombres_saisis += 1
    else:
        fin_saisie = True

if nb_nombres_saisis > 0:
    print("somme =", somme / nb_nombres_saisis)
else:
    print("aucun nombre saisi, donc pas de moyenne")
```

Exemple : calcul de racine carrée

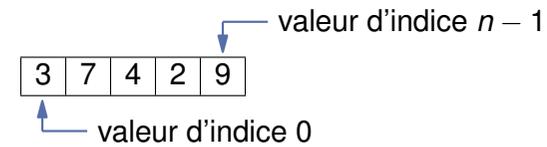
- ▶ $u_0 = 1, v_0 = x$
- ▶ $u_{n+1} = \frac{2}{\frac{1}{u_n} + \frac{1}{v_n}} \quad v_{n+1} = \frac{u_n + v_n}{2}$
- ▶ Convergence $\implies u_n \approx v_n \approx \sqrt{x}$

$u_n = u_1$
 $v_n = v_1$
 $u_{n_plus_1} = u_2$
 $v_{n_plus_1} = v_2$

```
def racine_carree(x) :  
    u_n = 1.0  
    v_n = float(x)  
    diff = 1.0 # initialisé pour rentrer dans la boucle  
    epsilon = 1e-3 # seuil pour déterminer s'il y a  
                  # convergence  
    while diff > epsilon :  
        u_n_plus_1 = 2 / ((1/u_n) + (1/v_n)) # calcule u_{n+1}  
        v_n_plus_1 = (u_n + v_n) / 2  
        if v_n_plus_1 > u_n_plus_1:  
            diff = v_n_plus_1 - u_n_plus_1  
        else:  
            diff = u_n_plus_1 - v_n_plus_1  
        u_n = u_n_plus_1 # à la prochaine itération,  
        v_n = v_n_plus_1 # u_{n_plus_1} deviendra un u_n  
    return u_n
```

Parcours de tableau de la gauche vers la droite

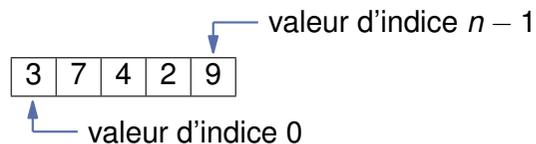
- ▶ Affichage du contenu d'un tableau 1 dimension :



```
def affichage(tableau, n):  
    """Affichage du contenu d'un tableau  
  
    tableau: le tableau 1 dimension  
    n : nombre d'éléments du tableau  
    """  
    i = 0 # indice du premier élément  
    while i < n:  
        print(tableau[i])  
        i += 1
```

Parcours de tableau de la droite vers la gauche

- ▶ Affichage du contenu d'un tableau de la droite vers la gauche :



```
def affichage_droite_gauche(tableau, n):  
    """Affichage du contenu d'un tableau  
  
    tableau: le tableau 1 dimension  
    n : nombre d'éléments du tableau  
    """  
    i = n-1 # indice du premier élément  
    while i >= 0:  
        print(tableau[i])  
        i -= 1
```

Affichage d'une partie de tableau

- ▶ Affichage des 4 éléments d'un tableau à partir de l'indice k



- ⚠ Ne pas dépasser la fin du tableau !

```
1 def affichage_4(tableau, n, k):  
2     i = 0 # le nombre de cellules déjà affichées  
3  
4     while k+i < n and i < 4:  
5         print(tableau[k+i])  
6         i += 1
```

- ▶ Ligne 4 :

- ▶ $k+i < n \implies$ cellule d'indice $k+i \in$ tableau
- ▶ $i < 4 \implies$ ne pas afficher plus de 4 éléments

Affichage d'une partie de tableau (autre direction)

- Affichage des 4 éléments d'un tableau à partir de l'indice k de la droite vers la gauche :



```
1 def affichage_4_droite_gauche(tableau, k):
2     i = 0 # le nombre de cellules déjà affichées
3
4     while k-i >= 0 and i < 4:
5         print(tableau[k-i])
6         i += 1
```

- Ligne 4 :

- $k-i \geq 0 \implies$ cellule d'indice $k-i \in$ tableau
- $i < 4 \implies$ ne pas afficher plus de 4 éléments

Affichage d'une partie de tableau (fin)

- Affichage de tous les éléments d'un tableau jusqu'à ce qu'on rencontre un nombre x :



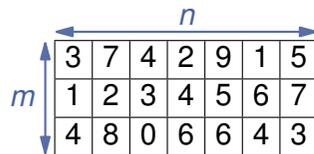
- ⚠ Ne pas dépasser la fin du tableau !

```
1 def affichage_x(tableau, n, x) :
2     i = 0
3
4     while i < n and tableau[i] != x :
5         print (tableau[i])
6         i += 1
```

- ⚠ Ligne 4 : l'ordre des tests est important !
Booléen évalué de la gauche vers la droite
 \implies Tester l'indice i avant d'accéder à la cellule d'indice i !

Affichage d'un tableau à 2 dimensions

- Affichage de tous les éléments d'un tableau 2D :



```
1 def affichage_x(tableau, m, n) :
2     y = 0 # indice des lignes
3
4     while y < m : # on parcourt toutes les lignes
5         x = 0 # indice de colonne
6         while x < n :
7             print(tableau[y,x])
8             x += 1
9         y += 1
```

- ⚠ Ligne 5 : à chaque ligne du tableau, remettre l'indice de colonne à 0 \implies doit être fait dans la boucle `while y < m`

Produit matrice vecteur 2D ?

- **Produit Matrice-Vecteur :**

$$\text{► } A = (a_{ij}), v = (v_j), b = Av = (b_i)$$

$$\text{► } b_i = \sum_j a_{ij} \times v_j$$

```
def prod_matrice_vecteur (mat, vect, nb_lignes, nb_cols):
    vect_produit = np.full(nb_lignes * nb_cols, 0)
    i = 0
    while i < nb_cols:
        j = 0
        while j < nb_lignes:
            vect_produit[i] += mat[i,j] * vect[i]
        return vect_produit
```

- Cette fonction est-elle correcte ?