

Algorithmique et programmation Python

Cours de maths - 1ère

Écrire un programme, ce n'est pas seulement « faire de l'informatique » : c'est dire à une machine, étape par étape, comment résoudre un problème de maths. En 1ère, tu réinvestis le langage Python vu en 2de pour travailler sur les **fonctions**, les **listes**, les **suites**, les équations (**dichotomie**), le hasard (**simulation**) et le **tri**. Chaque programme de ce chapitre a été réellement exécuté : les résultats affichés sont les vrais.

1. Variables, types et tests

Une **variable** est une case mémoire qui porte un nom et qui contient une valeur. On range une valeur dedans avec le signe `=`, qui est une *affectation* (et non une égalité mathématique).

Types et opérateurs à connaître

Les types usuels : **int** (entier), **float** (nombre décimal), **str** (texte), **bool** (**True** ou **False**), **list** (liste). Opérateurs : `+` `-` `*` `/`, puissance `**`, quotient entier `//` et reste `%`.

```
a = 17
b = 5
print(a // b, a % b, a / b, a ** 2)
```

Sortie obtenue :

```
3 2 3.4 289
```

Ici $17 = 5 \times 3 + 2$: le quotient entier vaut 3 et le reste 2. Le test de parité d'un entier n s'écrit donc **`n % 2 == 0`**.

Écrire un test

La structure **if ... elif ... else** exécute le premier bloc dont la condition est vraie. Les blocs sont *indentés* (décalés de 4 espaces) et la ligne du test se termine par deux-points. On combine les conditions avec **and**, **or**, **not**.

```
x = 7
if x % 2 == 0:
    print("pair")
elif x > 5:
    print("impair et grand")
else:
    print("impair et petit")
```

Pour $x = 7$: $7 \% 2$ vaut 1, donc le premier test est faux ; $7 > 5$ est vrai, donc le programme affiche « impair et grand ».

Deux pièges classiques

1) `=` affecte, `==` compare : écrire **if $x = 3$** est une erreur. 2) Les nombres décimaux sont stockés de façon approchée : en Python, $0.1 + 0.2 == 0.3$ est **False** car $0.1 + 0.2$ vaut 0.30000000000000004. On évite donc de comparer deux décimaux avec `==`.

2. Les fonctions

Une fonction Python est un petit programme réutilisable, avec des **paramètres** en entrée et une **valeur renvoyée** en sortie, comme une fonction mathématique.

Définition d'une fonction

On la définit avec **def**, on précise ses paramètres entre parenthèses, et on renvoie le résultat avec **return**. Dès que **return** est exécuté, la fonction s'arrête.

```
def f(x):
    return 2 * x ** 2 - 3 * x + 1

print(f(3))
print(f(-1))
```

Sortie obtenue :

```
10
6
```

Vérification à la main : $f(3) = 2 \times 9 - 9 + 1 = 10$ et $f(-1) = 2 + 3 + 1 = 6$. Une fonction peut contenir un test :

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

print n'est pas return

print affiche un texte à l'écran mais ne renvoie rien : on ne peut pas réutiliser le résultat. **return** renvoie une valeur utilisable ensuite (dans un calcul, un test, une autre fonction). Dans un exercice, on te demande presque toujours de *renvoyer* le résultat.

Les variables créées dans une fonction sont *locales* : elles n'existent que pendant son exécution. Une fonction peut aussi recevoir une autre fonction en paramètre (on s'en servira pour la dichotomie).

3. Les boucles

Boucle for et boucle while

Une boucle **for** répète un bloc un nombre de fois connu à l'avance. Une boucle **while** répète un bloc *tant qu'*une condition est vraie : on l'utilise quand on ne sait pas à l'avance combien de tours seront nécessaires.

Attention à **range** : **range(5)** donne 0, 1, 2, 3, 4 ; **range(2, 6)** donne 2, 3, 4, 5 (la borne de droite est exclue) ; **range(1, 10, 3)** donne 1, 4, 7 (le dernier nombre est le pas).

```
def somme(n):  
    s = 0  
    for k in range(1, n + 1):  
        s = s + k  
    return s  
  
print(somme(100))
```

Sortie obtenue :

5050

On retrouve la formule $1 + 2 + \dots + n = n(n+1)/2$: pour $n = 100$, $100 \times 101 / 2 = 5050$.

Voici une boucle while. On « déroule » le programme avec un **tableau de suivi** des variables :

```
x = 1
n = 0
while x < 50:
    x = 2 * x
    n = n + 1
```

Après le tour	0 (départ)	1	2	3	4	5	6
x	1	2	4	8	16	32	64
n	0	1	2	3	4	5	6

À la fin, $x = 64$ et $n = 6$: 64 est la première puissance de 2 qui dépasse 50.

Boucle infinie

Si la condition d'un while ne devient jamais fausse (on a oublié de modifier la variable testée), le programme ne s'arrête jamais. Vérifie toujours que quelque chose, dans la boucle, fait évoluer la condition.

4. Les listes

Liste

Une liste est une suite ordonnée de valeurs entre crochets, séparées par des virgules. Les éléments sont repérés par leur **indice**, qui commence à **0**. La longueur est donnée par **len(L)**.

```
L = [4, 9, 1, 7]
print(L[0], L[-1], len(L))
L.append(12)
L[1] = 0
print(L)
print(L[1:3])
```

Sortie obtenue :

```
4 7 4
[4, 0, 1, 7, 12]
```

[0, 1]

L[0] est le premier élément, **L[-1]** le dernier. **append** ajoute un élément à la fin. **L[1:3]** extrait les éléments d'indices 1 et 2 (le 3 est exclu). Une liste de n éléments a des indices de 0 à n – 1 : L[n] provoque une erreur.

Parcourir une liste

Deux façons : **for x in L** (on lit directement les valeurs) ou **for i in range(len(L))** (on utilise les indices, indispensable pour modifier la liste ou comparer des voisins).

```
L = [5, 8, 3]
for x in L:
    print(x * x)
M = [k ** 2 for k in range(5)]
print(M)
```

Sortie obtenue :

```
25
64
9
[0, 1, 4, 9, 16]
```

La deuxième construction, **[k ** 2 for k in range(5)]**, est une liste « en compréhension » : elle fabrique la liste des carrés de 0 à 4.

5. Parcours d'une liste : somme, maximum, recherche

Presque tous les algorithmes de ce chapitre suivent le même plan : on parcourt la liste en gardant à jour une variable qui résume ce qu'on a vu.

Algorithme du maximum

1) On suppose que le premier élément est le maximum provisoire. 2) On parcourt la liste : si un élément dépasse le maximum provisoire, il le remplace. 3) À la fin, le maximum provisoire est le maximum de la liste.

```
def maximum(L):
    m = L[0]
    for x in L:
```

```
    if x > m:
        m = x
    return m
```

Pour $L = [4, 9, 1, 7]$, m vaut successivement 4, 9, 9, 9 : le programme renvoie 9. Pour une liste de n éléments, il fait $n - 1$ comparaisons utiles.

Mauvaise initialisation

Initialiser $m = 0$ est faux si tous les éléments sont négatifs : le programme renverrait 0, qui n'est pas dans la liste. On initialise toujours avec **$L[0]$** .

La **moyenne** s'obtient en cumulant une somme dans une variable **s** initialisée à 0 puis en renvoyant **$s / \text{len}(L)$** . Pour **chercher** un élément, on arrête le parcours dès qu'on l'a trouvé :

```
def appartient(L, v):
    for x in L:
        if x == v:
            return True
    return False
```

Le **return False** est *en dehors* de la boucle : on ne peut conclure « absent » qu'après avoir tout examiné. Si on veut compter les éléments qui vérifient une condition, on utilise un compteur **k** initialisé à 0 et incrémenté (**$k = k + 1$**) à chaque élément qui convient.

6. Suites et algorithmes de seuil

Une suite définie par récurrence (par exemple $u_0 = 5$ et $u_{n+1} = 1,5 u_n - 2$) se calcule avec une boucle : on part de u_0 et on applique la relation n fois.

```
def terme(n):
    u = 5
    for k in range(n):
        u = 1.5 * u - 2
    return u

print(terme(0), terme(1), terme(2), terme(3))
```

Sortie obtenue :

5 5.5 6.25 7.375

À la main : $u_1 = 1,5 \times 5 - 2 = 5,5$ puis $u_2 = 1,5 \times 5,5 - 2 = 6,25$ puis $u_3 = 7,375$. La boucle **for** fait exactement n tours, donc après n tours u contient u_n .

Algorithme de seuil

Pour une suite croissante qui dépasse tout nombre, un **algorithme de seuil** renvoie le *plus petit entier* n tel que u_n dépasse un seuil S donné. On utilise un **while** avec un compteur n .

```
def seuil(S):  
    n = 0  
    u = 5  
    while u <= S:  
        u = 1.5 * u - 2  
        n = n + 1  
    return n
```

La condition du while est la *négation* de ce qu'on cherche : on continue tant que $u \leq S$, on s'arrête dès que $u > S$. Ici **seuil(1000)** renvoie 18 : c'est le premier rang pour lequel $u_n > 1000$.

Exemple : un capital qui double

On place 1000 € à 3 % par an. Au bout de combien d'années le capital dépasse-t-il 2000 € ? Le capital est multiplié par 1,03 chaque année.

```
c = 1000  
n = 0  
while c < 2000:  
    c = c * 1.03  
    n = n + 1  
print(n)
```

Le programme affiche 24 : après 23 ans on a $1000 \times 1,03^{23} \approx 1973,6$ €, après 24 ans $\approx 2032,8$ €. Réponse : 24 ans.

Mathbot vous conseille

Pour tester un algorithme de seuil, essaie un seuil facile à vérifier à la main (par

Fiche en ligne : maths-facile.eu/algorithme-et-programmation-python-cours-1ere/ · Cours, exercices et contrôles

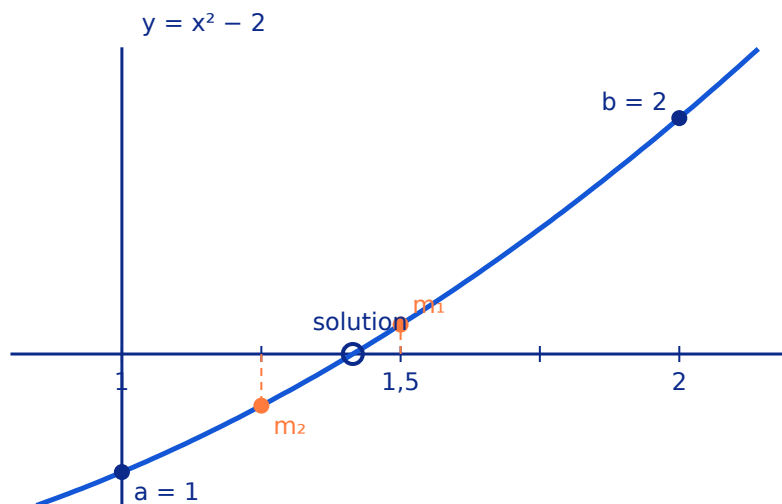
exemple $S = 6$ ici : $u_2 = 6,25$ est le premier terme supérieur à 6, donc le programme doit renvoyer 2).

7. Approximation d'une solution : la dichotomie

Principe de la dichotomie

Soit f une fonction continue sur $[a ; b]$ telle que $f(a)$ et $f(b)$ soient de signes contraires. Alors l'équation $f(x) = 0$ possède au moins une solution dans $[a ; b]$ (théorème des valeurs intermédiaires). Pour l'approcher, on coupe l'intervalle en deux au milieu m et on garde la moitié qui contient encore un changement de signe. À chaque étape, l'amplitude de l'intervalle est **divisée par 2**.

Exemple : $f(x) = x^2 - 2$ sur $[1 ; 2]$. On a $f(1) = -1 < 0$ et $f(2) = 2 > 0$. Le milieu est $m_1 = 1,5$ avec $f(1,5) = 0,25 > 0$: la solution est dans $[1 ; 1,5]$. Le milieu suivant est $1,25$ avec $f(1,25) = -0,4375 < 0$: la solution est dans $[1,25 ; 1,5]$. Sur la figure, on voit l'intervalle rétrécir autour de $\sqrt{2}$.



Programme de dichotomie

On répète tant que l'amplitude $b - a$ dépasse la précision voulue e . Si $f(a)$ et $f(m)$ sont de signes contraires (ou nuls), la solution est dans $[a ; m]$, sinon dans $[m ; b]$.

```
def dichotomie(f, a, b, e):
    while b - a > e:
        m = (a + b) / 2
        if f(a) * f(m) <= 0:
            b = m
```

```

        else:
            a = m
        return a, b

def g(x):
    return x ** 2 - 2

print(dichotomie(g, 1, 2, 0.01))

```

Sortie obtenue :

```
(1.4140625, 1.421875)
```

La solution $\sqrt{2} \approx 1,41421$ est bien dans l'intervalle renvoyé, dont l'amplitude est inférieure à 0,01.

Nombre d'étapes

Après n étapes l'amplitude vaut $(b - a) / 2^n$. Pour atteindre la précision e , il faut donc le plus petit entier n tel que $(b - a) / 2^n \leq e$. Avec $b - a = 1$ et $e = 0,01$: $2^7 = 128 \geq 100$, donc 7 étapes.

Hypothèses indispensables

La dichotomie suppose que f est continue et change de signe sur $[a ; b]$. Si ce n'est pas le cas, le programme tourne sans rien garantir. Pour garantir l'unicité de la solution, on ajoute que f est strictement monotone.

8. Simulation et hasard

Le module **random** fabrique des nombres pseudo-aléatoires. **randint(1, 6)** renvoie un entier au hasard entre 1 et 6 *inclus* (comme un dé) ; **random()** renvoie un décimal au hasard dans $[0 ; 1[$. À chaque exécution, les résultats changent.

```

from random import randint

def frequence_sept(N):
    c = 0
    for i in range(N):
        if randint(1, 6) + randint(1, 6) == 7:
            c = c + 1

```

```
return c / N

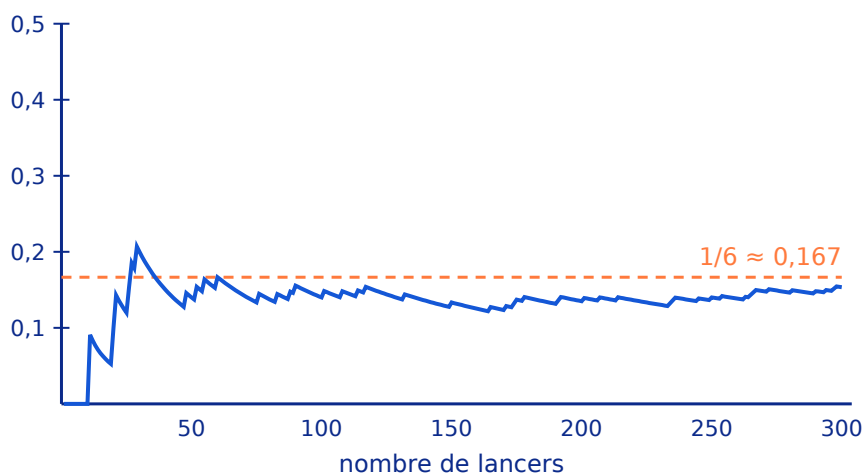
print(frequence_sept(10000))
```

Ce programme lance deux dés N fois et renvoie la **fréquence** des sommes égales à 7. La probabilité théorique est $6/36 = 1/6 \approx 0,167$ (6 couples sur 36 donnent 7). Le résultat affiché change à chaque exécution mais reste proche de 0,167 ; on obtient par exemple une valeur comprise entre 0,16 et 0,17.

Loi des grands nombres (admise)

Quand le nombre d'expériences N devient grand, la fréquence observée d'un événement se stabilise autour de sa probabilité. Pour N petit, les fluctuations sont grandes ; elles diminuent environ comme $1/\sqrt{N}$.

La figure montre une simulation de lancers de dé : la fréquence du 6 oscille beaucoup au début, puis se rapproche de $1/6$.



Une simulation n'est pas une preuve

Simuler donne une *estimation*, pas une valeur exacte. Deux exécutions donnent deux résultats différents : on ne conclut jamais « la probabilité vaut exactement 0,1673 » à partir d'une simulation.

9. Trier une liste

Trier, c'est ranger les éléments dans l'ordre croissant. Python sait le faire avec **sorted**, mais on étudie ici deux algorithmes simples pour comprendre comment ça marche.

Tri par sélection

Pour chaque position i , de gauche à droite : on cherche le plus petit élément parmi ceux d'indices i à $n - 1$, puis on l'échange avec l'élément d'indice i . Après le tour i , les cases 0 à i sont définitivement à leur place.

```
def tri_selection(L):
    n = len(L)
    for i in range(n - 1):
        imin = i
        for j in range(i + 1, n):
            if L[j] < L[imin]:
                imin = j
        L[i], L[imin] = L[imin], L[i]
    return L

print(tri_selection([5, 2, 9, 1]))
```

Sortie obtenue :

[1, 2, 5, 9]

Suivi sur [5, 2, 9, 1] : le tour 0 échange 5 et 1 (liste [1, 2, 9, 5]) ; le tour 1 laisse 2 à sa place ; le tour 2 échange 9 et 5 (liste [1, 2, 5, 9]).

départ	5	2	9	1
tour 0	1	2	9	5
tour 1	1	2	9	5
tour 2	1	2	5	9

orange : case déjà à sa place

Tri par insertion

On parcourt la liste de gauche à droite. À chaque tour, on prend l'élément x d'indice i et on le glisse à sa bonne place parmi les éléments déjà triés à sa gauche, en décalant d'une case vers la droite ceux qui sont plus grands.

```
def tri_insertion(L):
    for i in range(1, len(L)):
```

```

    x = L[i]
    j = i - 1
    while j >= 0 and L[j] > x:
        L[j + 1] = L[j]
        j = j - 1
    L[j + 1] = x
    return L

```

```
print(tri_insertion([5, 2, 9, 1]))
```

Sortie obtenue :

```
[1, 2, 5, 9]
```

Coût d'un tri

Le tri par sélection fait toujours $n(n - 1)/2$ comparaisons : 45 pour $n = 10$, mais 4950 pour $n = 100$. Quand la liste devient grande, ces tris simples deviennent lents (le coût est proportionnel à n^2).

Modification en place

Ces fonctions modifient la liste reçue : après **tri_selection(L)**, la liste L elle-même est triée. Pour garder l'original, on travaille sur une copie (**M = L[:]**).

À retenir

- = affecte, == compare ; // donne le quotient entier et % le reste.
- Une fonction se définit avec **def** et renvoie un résultat avec **return** (différent de **print**).
- **for** quand on connaît le nombre de tours, **while** tant qu'une condition est vraie ; **range(a, b)** exclut b.
- Dans une liste de n éléments, les indices vont de **0 à n - 1** ; **append** ajoute à la fin.
- Maximum, somme, comptage : un parcours avec une variable de résumé, correctement initialisée ($m = L[0]$, $s = 0$).
- Seuil : un **while** avec un compteur ; la condition est la négation de ce qu'on cherche.
- Dichotomie : f continue qui change de signe sur $[a ; b]$; l'amplitude est divisée par 2 à chaque étape, soit $(b - a)/2^n$ après n étapes.
- Simulation : la fréquence se stabilise autour de la probabilité quand N est grand, mais un résultat simulé reste approché.
- Tri par sélection : $n(n - 1)/2$ comparaisons ; tri par insertion : on glisse chaque

élément à sa place dans la partie déjà triée.