

Algorithmique et programmation Python

Cours de maths - Terminale

En Terminale, Python n'est plus seulement un outil pour « faire tourner » un calcul : il te sert à **explorer** une suite, à **approcher** la solution d'une équation, à **estimer** une intégrale, à **simuler** le hasard et à **compter** des cas. Dans ce chapitre, tu apprends les huit techniques que l'on retrouve toujours au bac, et surtout à prévoir ce que fait un programme avant de l'exécuter. Tous les programmes de ce cours ont été exécutés pour de vrai : les sorties affichées sont exactes (Python écrit les décimaux avec un point, nous les commentons avec une virgule).

1. Les bases à maîtriser

Quatre outils reviennent partout : la **variable**, la **boucle** (**for** quand on connaît le nombre de tours, **while** quand on s'arrête sur une condition), la **fonction** et la **liste**.

Fonction, liste, compréhension

Une fonction se définit par **def nom(paramètres):** et **renvoie** sa valeur avec **return**. Une liste se note entre crochets ; ses éléments sont numérotés **à partir de 0** ; **L[-1]** est le dernier. La *compréhension* **[expression for v in L if condition]** construit une nouvelle liste en une ligne.

```
def carre_plus(x, a):
    return x * x + a

L = [4, 9, 2, 7]
L.append(10)
total = 0
for v in L:
    total = total + v
print(carre_plus(3, 1), len(L), total, L[0], L[-1])
print([v * 2 for v in L if v > 5])
```

Sortie obtenue :

```
10 5 32 4 10
[18, 14, 20]
```

Lecture : **carre_plus(3, 1)** vaut $3 \times 3 + 1 = 10$; la liste a 5 éléments après le **append** ; la somme vaut $4 + 9 + 2 + 7 + 10 = 32$; enfin on garde les éléments strictement supérieurs à 5, puis on les double.

Pièges de base

range(5) donne 0, 1, 2, 3, 4 (la borne est *exclue*) ; **=** affecte, **==** compare ; **print** affiche mais ne renvoie rien, alors que **return** donne la valeur à la suite du programme ; tout ce qui est dans la boucle ou la fonction est *indenté* de la même façon.

2. Boucle while et seuil d'une suite

Une suite est définie par récurrence : on calcule ses termes un par un. Quand on cherche « le premier rang à partir duquel le terme dépasse (ou passe sous) un seuil », on ne sait pas à l'avance combien de tours faire : c'est le travail d'une boucle **while**.

Chercher le plus petit n tel que u_n franchit un seuil

1. Initialiser **u** au premier terme et un compteur **n = 0**.
2. Écrire la condition « on continue tant que le seuil n'est *pas encore* atteint ».
3. Dans la boucle : passer au terme suivant *puis* augmenter **n** de 1.
4. À la sortie, **n** est le plus petit rang cherché et **u** la valeur correspondante.

Exemple

On pose $u_0 = 500$ et $u_{n+1} = 0,85 u_n + 60$. On cherche le premier rang pour lequel $u_n \leq 420$.

```
u = 500
n = 0
while u > 420:
    u = 0.85 * u + 60
    n = n + 1
print(n, u)
```

Sortie obtenue :

```
10 419.6874404340722
```

Le programme répond $n = 10$: c'est le plus petit rang tel que $u_n \leq 420$, et on lit la valeur $u_{10} \approx 419,69$.

Contrôle par le calcul. Le point fixe ℓ vérifie $\ell = 0,85\ell + 60$, donc $\ell = 400$. La suite $v_n = u_n - 400$ est géométrique de raison 0,85 et de premier terme 100 : $u_n = 400 + 100 \times 0,85^n$. L'inéquation $u_n \leq 420$ équivaut à $0,85^n \leq 0,2$, soit $n \geq \ln(0,2) / \ln(0,85) \approx 9,9$, donc $n = 10$. Le programme et la théorie s'accordent.

Boucle infinie

Ici la suite décroît vers 400. Si tu avais écrit **while u > 390**, la condition ne deviendrait jamais fausse : le programme tournerait sans fin. Avant d'écrire la boucle, vérifie que le seuil est *réellement franchi* (limite de la suite, sens de variation).

Existence d'un seuil

Si (u_n) est croissante et tend vers $+\infty$, alors pour tout réel M il existe un rang N tel que $u_n > M$ pour tout $n \geq N$. Le programme à base de **while u <= M** s'arrête donc toujours. Pour une suite croissante qui converge vers ℓ , c'est impossible de dépasser un seuil supérieur à ℓ .

3. Listes et sommes de termes

Pour étudier le comportement d'une suite, on stocke souvent ses termes dans une liste (avec **append**) et on cumule leur somme. Le programme suivant génère les huit premiers termes de la suite géométrique $u_n = 3 \times (1/2)^n$, puis compare la somme obtenue à la formule $S = u_0 \times (1 - q^{n+1}) / (1 - q)$.

```
u = 3
termes = []
cumul = []
s = 0
for n in range(8):
    termes.append(u)
    s = s + u
    cumul.append(s)
    u = u / 2
print(termes)
print(cumul[-1], 6 * (1 - 0.5 ** 8))
```

Sortie obtenue :

```
[3, 1.5, 0.75, 0.375, 0.1875, 0.09375, 0.046875, 0.0234375]
5.9765625 5.9765625
```

Les deux valeurs de la dernière ligne sont égales : la formule $6 \times (1 - 0,5^8) = 5,976\dots$ donne bien le résultat de la boucle. On observe aussi que la somme se rapproche de 6, qui est la limite $u_0 / (1 - q) = 3 / 0,5$.

Mathbot conseille

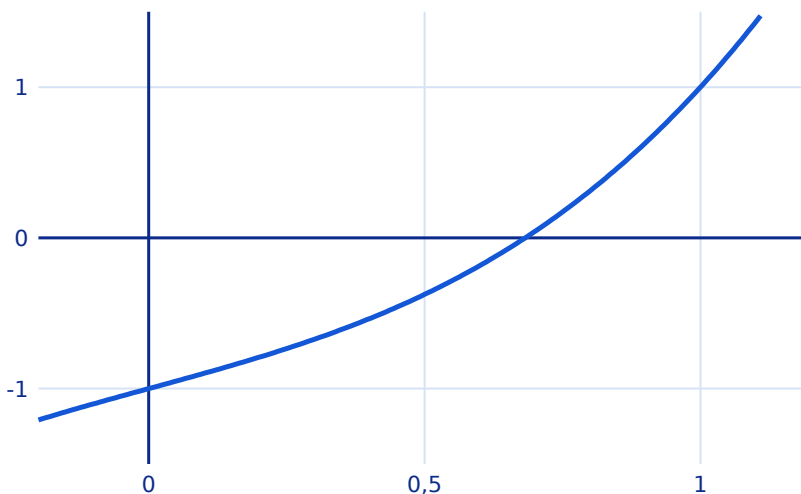
Pour vérifier un programme, calcule à la main les deux ou trois premiers termes et compare. Si ça ne colle pas, regarde l'ordre des instructions dans la boucle : on ajoute le terme *avant* de passer au suivant.

4. Dichotomie : approcher la solution d'une équation

Théorème des valeurs intermédiaires (rappel)

Si f est continue sur $[a ; b]$ et si $f(a)$ et $f(b)$ sont de signes contraires, alors l'équation $f(x) = 0$ a au moins une solution dans $[a ; b]$. Si de plus f est strictement monotone, cette solution est **unique**.

Principe de la dichotomie : on coupe $[a ; b]$ en deux au milieu $m = (a + b) / 2$, et on garde la moitié où la fonction change de signe. À chaque étape, la longueur de l'intervalle est divisée par 2.



Écrire une dichotomie

1. Vérifier que f est continue, strictement monotone et que $f(a) \times f(b) \leq 0$.
2. Tant que $b - a$ est strictement supérieur à la précision e : calculer m .
3. Si $f(a) \times f(m) \leq 0$, la solution est dans $[a ; m]$, donc **$b = m$** ; sinon elle est dans $[m ; b]$, donc **$a = m$** .
4. À la fin, a et b encadrent la solution avec une amplitude inférieure ou égale à e .

Exemple

$f(x) = x^3 + x - 1$ est continue et strictement croissante sur $[0 ; 1]$ ($f'(x) = 3x^2 + 1 > 0$), avec $f(0) = -1$ et $f(1) = 1$: l'équation $f(x) = 0$ a une unique solution α dans $[0 ; 1]$.

```
def f(x):  
    return x ** 3 + x - 1  
  
def dichotomie(f, a, b, e):  
    etapes = 0  
    while b - a > e:  
        m = (a + b) / 2  
        if f(a) * f(m) <= 0:  
            b = m  
        else:  
            a = m  
        etapes = etapes + 1  
    return a, b, etapes  
  
print(dichotomie(f, 0, 1, 0.001))
```

Sortie obtenue :

```
(0.681640625, 0.6826171875, 10)
```

Le programme renvoie un encadrement de α d'amplitude inférieure à 0,001, obtenu en 10 étapes (la dernière valeur de la sortie est le nombre d'étapes).

Nombre d'étapes

Après k étapes, l'amplitude vaut $(b - a) / 2^k$. Pour atteindre la précision e , il faut donc $2^k \geq (b - a) / e$, c'est-à-dire $k \geq \log_2((b - a) / e)$. Ici, $(b - a) / e = 1000$ et $2^{10} = 1024 > 1000$: 10 étapes suffisent, et 9 ne suffisent pas ($2^9 = 512$).

5. Méthode de Newton

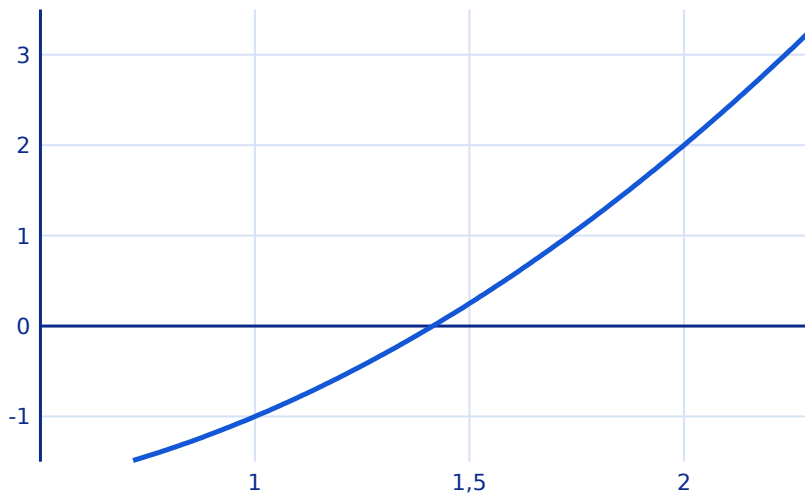
Pour une fonction f dérivable, la **méthode de Newton** remplace la courbe par sa tangente. On part d'une valeur x_0 proche de la solution, on trace la tangente en x_0 et on prend son point d'intersection avec l'axe des abscisses : c'est x_1 . On recommence.

Formule de récurrence

Fiche en ligne : maths-facile.eu/algorithmique-et-programmation-python-cours-terminale/ · Cours, exercices et contrôles corrigés sur maths-facile.eu · page 5/12

La tangente en x_n a pour équation $y = f(x_n) + f'(x_n) \times (x - x_n)$. Elle coupe l'axe des abscisses lorsque $y = 0$, donc en

$x_{n+1} = x_n - f(x_n) / f'(x_n)$ (à condition que $f'(x_n) \neq 0$).



Racine carrée de 2

Avec $f(x) = x^2 - 2$ et $f'(x) = 2x$, la formule devient $x_{n+1} = x_n - (x_n^2 - 2) / (2x_n) = (x_n + 2 / x_n) / 2$. Partons de $x_0 = 1$.

```
def newton(f, fp, x0, n):
    x = x0
    for i in range(n):
        x = x - f(x) / fp(x)
        print(i + 1, x)
    return x

newton(lambda x: x * x - 2, lambda x: 2 * x, 1, 5)
```

Sortie obtenue :

```
1 1.5
2 1.4166666666666667
3 1.4142156862745099
4 1.4142135623746899
5 1.4142135623730951
```

En cinq tours, on dispose de la valeur de $\sqrt{2}$ avec une quinzaine de décimales justes.

Remarque le « miracle » : le nombre de décimales exactes **double à peu près à chaque étape**, alors que la dichotomie n'en gagne qu'environ 0,3 par étape (elle divise par 2 la

Fiche en ligne : maths-facile.eu/algorithmique-et-programmation-python-cours-terminale/ · Cours, exercices et

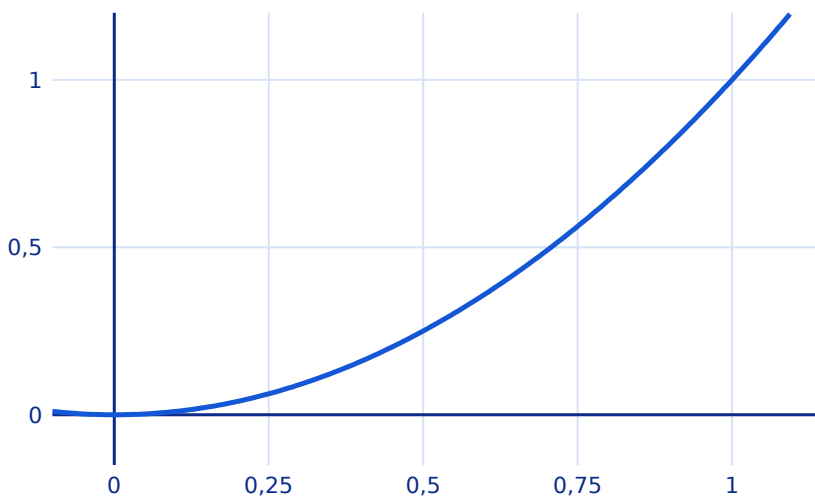
longueur de l'intervalle).

Limites de Newton

Il faut que $f'(x_n)$ ne soit pas nul (tangente horizontale = pas d'intersection). Un mauvais x_0 peut faire diverger la suite ou l'envoyer vers une autre solution. La méthode ne fournit pas d'encadrement garanti, contrairement à la dichotomie : on la complète souvent par un contrôle du signe de f .

6. Intégrale par la méthode des rectangles

Pour une fonction f **continue, positive et croissante** sur $[a ; b]$, l'intégrale de f entre a et b est l'aire sous la courbe. On découpe $[a ; b]$ en n bandes de largeur $h = (b - a) / n$ et on encadre l'aire par deux sommes de rectangles.



Encadrement par les rectangles

Pour f croissante, avec $x_k = a + k \times h$:

$h \times [f(x_0) + f(x_1) + \dots + f(x_{n-1})] \leq \int f(x) dx \leq h \times [f(x_1) + \dots + f(x_n)]$ (intégrale de a à b).

L'écart entre les deux sommes est exactement $h \times (f(b) - f(a))$: il tend vers 0 quand n tend vers $+\infty$. Si f est décroissante, les deux membres de l'encadrement s'échangent.

Avec $f(x) = x^2$ sur $[0 ; 1]$ (dont l'intégrale vaut $1/3$), voici ce que donne le programme pour $n = 4$ puis $n = 1000$.

```
def rectangles(f, a, b, n):  
    h = (b - a) / n  
    bas = 0  
    haut = 0  
    for k in range(n):
```

```

        bas = bas + f(a + k * h)
        haut = haut + f(a + (k + 1) * h)
    return h * bas, h * haut

f = lambda x: x ** 2
print(rectangles(f, 0, 1, 4))
print(rectangles(f, 0, 1, 1000))

```

Sortie obtenue :

```

(0.21875, 0.46875)
(0.33283349999999995, 0.33383349999999995)

```

Pour $n = 4$, on a bien $0,21875 \leq 1/3 \leq 0,46875$, avec un écart de $0,25 = (1/4) \times (1 - 0)$.
 Pour $n = 1000$, l'encadrement est serré à 0,001 près : $1/3 \approx 0,3333$ se trouve entre 0,33283 et 0,33383.

Bien repérer les indices

Dans le programme, **range(n)** parcourt $k = 0, 1, \dots, n-1$: la somme « du bas » utilise **f(a + k*h)** (rectangles sous la courbe pour une fonction croissante) et la somme « du haut » utilise **f(a + (k+1)*h)**. Si f est décroissante, l'ordre est inversé.

7. Simulations aléatoires

Le module **random** fournit des nombres pseudo-aléatoires : **random.random()** donne un réel dans $[0 ; 1[$ (loi uniforme), **random.randint(a, b)** un entier entre a et b *inclus*. Avec **random.seed(n)**, on fixe la « graine » : le programme redonne exactement les mêmes résultats à chaque exécution, ce qui permet de tester et de comparer.

Loi des grands nombres

Lorsqu'on répète un grand nombre N de fois, de façon indépendante, une même expérience, la fréquence d'un événement se rapproche de sa probabilité ; la fluctuation est de l'ordre de $1 / \sqrt{N}$. Simuler 100 fois plus d'expériences gagne donc *une seule* décimale de précision.

```

import random
random.seed(12)
N = 10000
six = 0

```

```
for i in range(N):
    if random.randint(1, 6) == 6:
        six = six + 1
print(six, six / N)
```

Sortie obtenue :

```
1674 0.1674
```

Le « six » apparaît environ 1 fois sur 6 : on obtient la fréquence ci-dessus, proche de $1/6 \approx 0,1667$.

Estimer une probabilité par simulation

1. Écrire une fonction qui simule *une* expérience et renvoie le résultat voulu (succès ou non).
2. La répéter N fois dans une boucle et compter les succès.
3. Calculer la fréquence **succes / N** et la comparer à la valeur théorique.

On peut aussi simuler une variable X qui suit la loi binomiale $B(n ; p)$: on répète n épreuves de Bernoulli et on compte les succès. Le programme ci-dessous estime $P(X = 3)$ pour $B(10 ; 0,3)$ et la compare à la valeur exacte $C(10, 3) \times 0,3^3 \times 0,7^7$.

```
import random
from math import comb
random.seed(5)

def succes(n, p):
    return sum(1 for i in range(n) if random.random() < p)

essais = 20000
compte = 0
for t in range(essais):
    if succes(10, 0.3) == 3:
        compte = compte + 1
print(compte / essais, comb(10, 3) * 0.3 ** 3 * 0.7 ** 7)
```

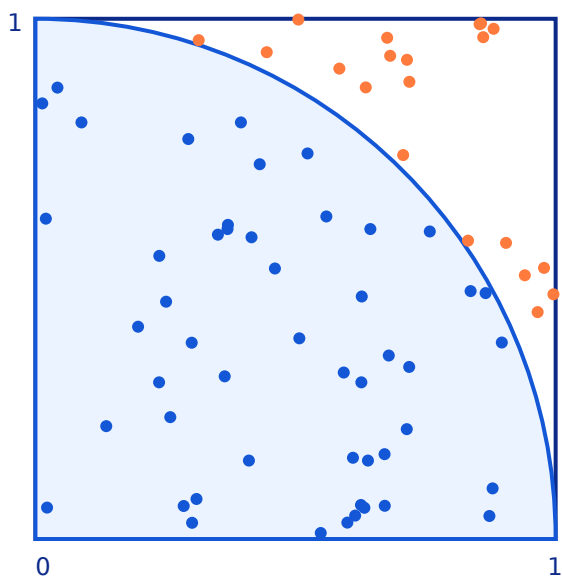
Sortie obtenue :

```
0.267 0.2668279319999998
```

La fréquence simulée (premier nombre) est proche de la valeur exacte $\approx 0,2668$ (second nombre). Les deux nombres ne sont *pas* égaux : une simulation donne une estimation, jamais une valeur exacte.

Méthode de Monte-Carlo

On tire au hasard un point $(x ; y)$ dans le carré $[0 ; 1] \times [0 ; 1]$; il tombe dans le quart de disque si $x^2 + y^2 \leq 1$. Cette probabilité est $(\pi / 4) / 1$, donc $\pi \approx 4 \times (\text{fréquence})$. Sur la figure, 70 points : les bleus sont dans le quart de disque.



8. Dénombrement par programme

Pour dénombrer, on peut faire parcourir tous les cas par le programme et compter ceux qui conviennent : une boucle **for** par « choix » (boucles imbriquées). Le module **math** donne directement les coefficients binomiaux avec **comb(n, k)**, et le module **itertools** génère les arrangements (**permutations(L, k)**) et les combinaisons (**combinations(L, k)**).

```
from math import comb
import itertools

# couples (a, b) de deux des dont la somme vaut 8
nb = 0
for a in range(1, 7):
    for b in range(1, 7):
        if a + b == 8:
            nb = nb + 1
print(nb)
```

```
print(comb(6, 2), len(list(itertools.permutations("ABCD", 3))))
```

Sortie obtenue :

```
5
15 24
```

Il y a 5 couples de somme 8 : (2 ; 6), (3 ; 5), (4 ; 4), (5 ; 3), (6 ; 2). Ensuite, $\text{comb}(6, 2) = 15$ sous-ensembles de 2 éléments parmi 6 et il y a $4 \times 3 \times 2 = 24$ mots de 3 lettres distinctes avec les lettres A, B, C, D.

Rappels de dénombrement

k-uplets d'un ensemble à n éléments : n^k ; arrangements (ordre compté, sans répétition) : $n \times (n-1) \times \dots \times (n-k+1)$; combinaisons : $C(n, k) = n! / (k! \times (n-k)!)$, avec $C(n, k) = C(n-1, k-1) + C(n-1, k)$.

Cette dernière relation (formule de Pascal) permet de construire le triangle de Pascal sans factorielle : chaque ligne s'obtient en ajoutant deux à deux les termes de la ligne précédente.

```
def pascal(n):
    ligne = [1]
    for i in range(n):
        ligne = [1] + [ligne[j] + ligne[j + 1] for j in
range(len(ligne) - 1)] + [1]
    return ligne

print(pascal(5))
print(pascal(6))
```

Sortie obtenue :

```
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
```

La ligne de rang 5 est 1, 5, 10, 10, 5, 1 : on y lit $C(5, 2) = 10$.

9. Écrire un programme fiable

Avant de lancer un programme

1. Écrire sur papier ce que la fonction doit renvoyer pour un petit exemple ($n = 1$, $n = 2$).
2. Choisir la structure : **for** si le nombre de tours est connu, **while** si on s'arrête sur une condition.
3. Tester les cas limites : liste vide, $n = 0$, seuil déjà franchi.
4. Comparer la sortie à un calcul exact quand on le connaît.

Erreurs classiques

Oublier de mettre à jour la variable de la condition d'un **while** (boucle infinie) ; confondre / (division décimale) et // (quotient entier) ; écrire $x \wedge 2$ au lieu de $x ** 2$ (en Python, $\wedge$ est un « ou exclusif ») ; tester l'égalité de deux décimaux avec `==` (utilise plutôt `abs(a - b) < 1e-9`) ; confondre l'indice (qui commence à 0) avec le rang du terme.

À retenir

- **for** : nombre de tours connu ; **while** : arrêt sur une condition. Un seuil de suite se programme avec un **while**, et le compteur final donne le plus petit rang.
- Une fonction **renvoie** avec **return** ; **print** ne fait qu'afficher. Les listes commencent à l'indice 0 et **range(n)** exclut n .
- Dichotomie : f continue et changement de signe ; l'amplitude est divisée par 2 à chaque étape ; il faut $k \geq \log_2((b - a) / \epsilon)$ étapes.
- Newton : $x_{n+1} = x_n - f(x_n) / f'(x_n)$; convergence très rapide mais sans garantie (x_0 bien choisi, f' non nul).
- Rectangles : $h = (b - a) / n$; pour f croissante, somme du bas $\leq$ intégrale $\leq$ somme du haut ; l'écart vaut $h \times (f(b) - f(a))$.
- Simulation : fréquence sur N répétitions $\approx$ probabilité, avec une fluctuation en $1 / \sqrt{N}$; **random.seed** fixe le résultat.
- Dénombrement : boucles imbriquées, **comb(n, k)**, **itertools** ; relation de Pascal $C(n, k) = C(n-1, k-1) + C(n-1, k)$.