

Premiers pas : variables, conditions et boucles

Chapitre 32

Julien BÉRARD

Lycée Saint-Augustin
Tale — Python pour les mathématiques

2026–2027

Plan du chapitre

- 1 L'environnement de travail : console et script
- 2 Variables, affectation et types
- 3 Opérations arithmétiques
- 4 Afficher et saisir
- 5 Booléens et logique
- 6 L'instruction conditionnelle
- 7 La boucle `while`
- 8 La boucle `for` et `range`
- 9 Lire un message d'erreur

- Chacun a rencontré Python en Seconde et en Première, à des degrés très divers : on **repart de zéro**, rapidement et solidement.
- Les briques dont tout le reste dépend :
 - les **variables**, les **types** de valeurs, les **opérations** ;
 - les **booléens** et l'**instruction conditionnelle** ;
 - les deux boucles, **while** et **for**.
- Python, **laboratoire des mathématiques du moment** :
 - *et, ou, non* deviennent **and**, **or**, **not** ; une boucle vérifie les lois de De Morgan ;
 - la division euclidienne se lit avec **//** et **%** ; une boucle dresse les diviseurs d'un entier.
- On travaille **en ligne**, sans rien installer : Basthon (basthon.fr) ou Capytale.
- Tous les exemples donnent le code et la **sortie exacte** : reproduisez-les, modifiez-les, provoquez des erreurs !

Définition : console et script

- La **console** exécute une instruction à la fois : après l'invite `>>>`, elle affiche la valeur de l'expression tapée. L'outil de l'essai rapide.
- Un **script** est un fichier d'instructions exécutées de haut en bas. **Rien ne s'affiche**, sauf ce que l'on demande avec `print`.

Console

```
>>> 2 ** 10
1024
>>> print("Bonjour")
Bonjour
```

Script

```
1 2 + 3
2 print(7 * 8)
3 x = 2 ** 10
4 print(x)
```

Sortie :

```
56
1024
```

Ligne 1 : 5 est calculé mais **perdu**, ni affiché ni rangé dans une variable.

Définition : variable et affectation

Une **variable** est un nom qui désigne une valeur rangée en mémoire. L'affectation `nom = expression` évalue d'abord la partie droite, puis range la valeur sous le nom de gauche ; l'ancienne valeur est oubliée.

```
>>> a = 5
>>> a = a + 1
>>> a
6
>>> b = a * 2
>>> b
12
```

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

- `a = a + 1` est une **instruction**, pas une équation.
- L'**affectation simultanée** évalue toute la droite d'abord : elle échange deux valeurs.
- Noms sans espace ni accent ; `somme` $\neq$ `Somme`.

Définition : types

Toute valeur a un **type**, renvoyé par la fonction `type` : `int` (entiers, sans limite de taille), `float` (flottants, écrits avec un point : `0.5`, `1e-9`), `bool` (`True`, `False`), `str` (chaînes, entre `"..."` ou `'...'`). Les fonctions `int`, `float`, `str` **convertissent** lorsque c'est possible.

```
>>> type(7.0)
<class 'float'>
>>> type(7 > 3)
<class 'bool'>
>>> int(7.9)
7
>>> str(7) + "1"
'71'
```

- `int(7.9)` **tronque**, il n'arrondit pas.
- `+` entre deux chaînes les met bout à bout : `"7" + "1"` vaut `"71"`, pas 8.
- `int("sept")` provoque une `ValueError`.

Proposition : opérateurs

+ - * somme, différence, produit
/ division, **toujours flottante**
// quotient euclidien : 17 // 5 vaut 3
% reste euclidien : 17 % 5 vaut 2
** puissance : 2 ** 10 vaut 1024

Priorités des mathématiques : **, puis * / // %, puis + - ; à égalité, de gauche à droite, **sauf ****, de droite à gauche.

```
>>> 2 + 3 * 4
```

```
14
```

```
>>> -2 ** 2
```

```
-4
```

```
>>> (-2) ** 2
```

```
4
```

```
>>> 2 ** 3 ** 2
```

```
512
```

-2 ** 2 vaut $-(2^2)$ et 2 ** 3 ** 2 vaut $2^{(3^2)}$. **Dans le doute, des parenthèses.**

Division euclidienne

Exemple : $a = bq + r$ avec $0 \leq r < b$

```
1 a = 47
2 b = 6
3 q = a // b
4 r = a % b
5 print(f"{a} = {b} x {q} + {r}")
6 print(0 <= r < b)
```

Sortie :

```
47 = 6 x 7 + 5
True
```

Dividende négatif : reste **positif**, comme dans le cours.

```
>>> -17 // 5
-4
>>> -17 % 5
3
```

En effet $-17 = 5 \times (-4) + 3$.

À retenir

b divise a si, et seulement si, $a \% b == 0$: c'est le test de divisibilité de tout le livre.

Les flottants ne sont pas les réels

Remarque

Un flottant s'écrit en base 2 avec un nombre fini de chiffres : ce n'est pas le cas de 0,1 ni de 0,2. Leur représentation est **arrondie** et l'addition propage l'arrondi.

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
>>> abs((0.1 + 0.2) - 0.3) < 1e-9
True
>>> 6 / 2
3.0
```

1. On ne teste **jamaïs** l'égalité de deux flottants avec `==` : on teste que leur écart est inférieur à une tolérance.
2. Quand le calcul est entier, on reste chez les entiers : `//` et `%` plutôt que `/`.

Retour détaillé au chapitre Suites, équations et nombres flottants.

Affichage

`print` affiche ses arguments séparés d'un espace. Dans une **f-string** `f"..."`, toute expression entre accolades est remplacée par sa valeur ; `{x:.4f}` donne quatre décimales.

```
>>> n = 12
>>> print("n =", n)
n = 12
>>> print(f"n**2 = {n**2}")
n**2 = 144
>>> print(f"{2/3:.4f}")
0.6667
```

Saisie

`input` affiche un message et renvoie la réponse **sous forme de chaîne** : il faut convertir avec `int` ou `float`.

```
1 age = int(input("Votre age ? "))
2 print(f"Dans 10 ans : {age + 10}")
```

Sortie : (en tapant 17)

Votre age ? 17

Dans 10 ans : 27

Dans la suite, on fixe les données au début du script (`n = 2025`).

Comparaisons : = et ==

Définition : comparaisons

Une **comparaison** est une expression booléenne : `==`, `!=`, `<`, `<=`, `>`, `>=`.

Elles sont moins prioritaires que l'arithmétique : `n % 2 == 0` se lit `(n % 2) == 0`.

```
>>> x = 4
>>> x == 4
True
>>> 3 != 5
True
>>> 0 < x < 10
True
```

Ne pas confondre

- `=` : **affectation**, une instruction qui modifie une variable ;
- `==` : **test d'égalité**, une expression qui vaut `True` ou `False` et ne modifie rien.

Les comparaisons s'enchaînent : `0 < x < 10` a le sens mathématique de $0 < x < 10$.

Définition : and, or, not

- `P and Q` vaut `True` si, et seulement si, `P` et `Q` valent tous deux `True` ;
- `P or Q` vaut `True` si, et seulement si, l'un au moins vaut `True` (ou inclusif) ;
- `not P` vaut `True` si, et seulement si, `P` vaut `False`.

Priorité décroissante : `not`, `and`, `or`, tous moins prioritaires que les comparaisons : `n % 2 == 0 and n % 3 == 0` teste la divisibilité par 6.

Remarque

- Pas d'opérateur pour $P \implies Q$: on écrit sa définition, `(not P) or Q`.
- Les booléens sont des entiers déguisés : `True + True` vaut 2. On compte ainsi des cas favorables en additionnant des tests.

Tables de vérité et lois de De Morgan

Tables de $P \wedge Q$ et $P \vee Q$

```
1 for P in (True, False) :
2     for Q in (True, False) :
3         print(P, Q, P and Q, P or Q)
```

Sortie :

```
True True True True
True False False True
False True False True
False False False False
```

De Morgan : égalité des deux membres

```
1 for P in (True, False) :
2     for Q in (True, False) :
3         print(P, Q,
4             (not (P and Q)) == ((not P) or (not Q)),
5             (not (P or Q)) == ((not P) and (not Q)))
```

Sortie :

```
True True True True
True False True True
False True True True
False False True True
```

Que des `True` dans les deux dernières colonnes : une table de vérité est une **vérification exhaustive**, donc une démonstration. Les parenthèses autour de `not (P and Q)` sont indispensables : `==` est prioritaire sur `not`.

if, elif, else

```
if condition1 :  
    bloc 1  
elif condition2 :  
    bloc 2  
else :  
    bloc 3
```

Définition : instruction conditionnelle

Si `condition1` vaut `True`, le bloc 1 est exécuté et le reste est ignoré ; sinon `condition2` est évaluée, et ainsi de suite. Le bloc du `else` n'est exécuté que si **toutes** les conditions sont fausses. `elif` (plusieurs possibles) et `else` sont facultatifs.

L'indentation fait partie de la syntaxe

Un **bloc** est un groupe de lignes décalées de **quatre espaces** ; il se termine à la première ligne qui revient à l'alignement précédent. Deux `if` successifs sont indépendants ; un `if` suivi d'un `else` forme une alternative.

Deux exemples : parité, signe d'un réel

Exemple : parité

```
1 n = 17
2 if n % 2 == 0 :
3     print(n, "est pair")
4 else :
5     print(n, "est impair")
```

Sortie :

```
17 est impair
```

Exemple : signe d'un réel

```
1 x = -3
2 if x > 0 :
3     print("strictement positif")
4 elif x == 0 :
5     print("nul")
6 else :
7     print("strictement négatif")
```

Sortie :

```
strictement négatif
```

Les trois cas s'excluent et le `else` final recueille tout le reste : c'est la **disjonction des cas** du chapitre Divers modes de raisonnement.

Boucle conditionnelle : recherche d'un diviseur

```
while condition :  
    bloc
```

Définition : boucle while

Le bloc est exécuté **tant que** la condition est vraie ; elle est réévaluée avant chaque tour. Le nombre de tours est **inconnu à l'avance**.

Exemple : plus petit diviseur $d \geq 2$ de $n \geq 2$

```
1 n = 91  
2 d = 2  
3 while n % d != 0 :  
4     d = d + 1  
5 print("plus petit diviseur de", n, ":", d)
```

Sortie :

```
plus petit diviseur de 91 : 7
```

La boucle **se termine toujours** : n est divisible par lui-même, donc au pire d atteint n , et n est premier. Le chapitre Nombres premiers s'arrêtera dès que $d^2 > n$.

L'algorithme de seuil

Exemple : $u_0 = 1$, $u_{n+1} = 2u_n$. À partir de quel rang $u_n \geq 1000$?

```
1 u = 1
2 n = 0
3 while u < 1000 :
4     u = 2 * u
5     n = n + 1
6 print(n, u)
```

Sortie :

```
10 1024
```

- Les deux variables **avancent ensemble** : à chaque tour, **u** contient u_n pour la valeur courante de **n**.
- À la sortie, **u** contient la **première** valeur qui rend la condition fausse : $u_{10} = 1024 \geq 1000$.
- Retenir ce décalage d'un tour évite bien des erreurs.

Remarque

C'est l'**algorithme de seuil**, que le chapitre Suites justifiera par la définition d'une limite infinie.

Attention

Si la condition ne devient jamais fausse, la boucle ne s'arrête pas : bouton *Stop* de Basthon. Deux précautions :

1. le bloc doit modifier une variable de la condition (oublier `n = n + 1` est l'erreur classique) ;
2. pendant la mise au point, un **compteur de tours** et une **borne** dans la condition.

```
1 u = 1
2 n = 0
3 while u != 0 and n < 100 :
4     u = u / 2
5     n = n + 1
6 print(n, u)
```

Sortie :

```
100 7.888609052210118e-31
```

- Mathématiquement, $u = 2^{-n}$ n'est jamais nul : sans borne, la boucle ne s'arrêterait pas.
- Arrêt après 100 tours : $2^{-100} \simeq 7,9 \times 10^{-31}$ (**e-31** signifie $\times 10^{-31}$).
- Avec la borne `n < 2000`, arrêt après 1075 tours avec `u` égal à **0.0** : la machine a arrondi à zéro.

Définition : boucle for

`for k in range(a, b)` : exécute le bloc pour $k = a, a + 1, \dots, b - 1$. La variable `k` avance **seule** : on ne l'incrmente pas. Le nombre de tours, $b - a$, est **connu d'avance**.

Proposition : les trois formes de range

- `range(b)` : $0, 1, \dots, b - 1$ (b valeurs) ;
- `range(a, b)` : $a, a + 1, \dots, b - 1$;
- `range(a, b, p)` : $a, a + p, a + 2p, \dots$ strictement avant b (après b si $p < 0$).

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(2, 7))
[2, 3, 4, 5, 6]
>>> list(range(0, 20, 5))
[0, 5, 10, 15]
>>> list(range(10, 0, -3))
[10, 7, 4, 1]
```

Borne de droite exclue : pour $\llbracket 1, n \rrbracket$, on écrit `range(1, n+1)`.

Définition : accumulateur

Variable **initialisée avant la boucle** et mise à jour à chaque tour, qui contient à la sortie le résultat cherché.

résultat	avant la boucle	à chaque tour
somme	<code>somme = 0</code>	<code>somme = somme + terme</code>
produit	<code>produit = 1</code>	<code>produit = produit * terme</code>
compteur	<code>compteur = 0</code>	<code>compteur = compteur + 1</code> si condition
maximum	<code>maximum = terme initial</code>	<code>maximum = terme</code> si <code>terme > maximum</code>

À retenir

L'initialisation est l'**élément neutre** de l'opération : 0 pour la somme, 1 pour le produit.

Raccourci : `somme += k` équivaut à `somme = somme + k`; de même `*=`, `-=`.

Exemple : vérifier deux formules du chapitre Réurrence

On confronte $\sum_{k=1}^n k = \frac{n(n+1)}{2}$ et $\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$ au calcul terme à terme.

```
1 n = 100
2 somme = 0
3 carres = 0
4 for k in range(1, n + 1) :
5     somme = somme + k
6     carres = carres + k ** 2
7 print(somme, n * (n + 1) // 2)
8 print(carres, n * (n + 1) * (2 * n + 1) // 6)
```

Sortie :

```
5050 5050
338350 338350
```

- L'ordinateur **vérifie** la formule pour une valeur de n , il ne la **démontre** pas : la démonstration est la récurrence du cours.
- `//` garde le calcul chez les entiers, la division étant exacte.

Produit, compteur, maximum

20!

```
1 produit = 1
2 for k in range(1, 21) :
3     produit = produit * k
4 print(produit)
```

Sortie :

2432902008176640000

Multiples de 3 dans $\llbracket 1, 100 \rrbracket$

```
1 compteur = 0
2 for k in range(1, 101) :
3     if k % 3 == 0 :
4         compteur += 1
5 print(compteur)
```

Sortie :

33

Max des restes de k^2 par 23

```
1 maximum = 0
2 for k in range(1, 21) :
3     r = (k * k) % 23
4     if r > maximum :
5         maximum = r
6 print(maximum)
```

Sortie :

18

- Le compteur ne progresse que dans les tours où la condition est vraie : $33 = \lfloor 100/3 \rfloor$.
- Maximum initialisé à 0 car tous les restes sont positifs ; en général, on l'initialise avec le **premier terme examiné**.

Termes d'une suite arithmétique, d'une suite géométrique

Exemple : $u_0 = 3$, raison 4 ; $v_0 = 5$, raison 1.5

```
1 u = 3          # u_0
2 v = 5          # v_0
3 for n in range(1, 6) :
4     u = u + 4    # u_n = u_{n-1} + 4
5     v = v * 1.5  # v_n = 1,5 v_{n-1}
6     print(n, u, v)
```

Sortie :

```
1 7 7.5
2 11 11.25
3 15 16.875
4 19 25.3125
5 23 37.96875
```

- À chaque tour, **u** passe de u_{n-1} à u_n , et **v** de v_{n-1} à v_n .
- On vérifie $u_5 = 3 + 5 \times 4 = 23$ et $v_5 = 5 \times 1,5^5 = 37,96875$.
- Ce schéma, **une variable mise à jour à chaque tour**, est celui de toutes les suites récurrentes.

Lire un message d'erreur (1)

Python indique le **type** de l'erreur et la **ligne**. On lit toujours la **dernière ligne** d'abord.

SyntaxError : pas du Python

```
1 print("Bonjour")
2 print("Au revoir")
```

```
File "script.py", line 1
    print("Bonjour")
    ^
```

SyntaxError: '(' was never closed

NameError : nom inconnu

```
1 somme = 0
2 for k in range(1, 4) :
3     somme = somme + k
4 print(somme)
```

Traceback (most recent call last):

```
File "script.py", line 4, in <module>
    print(somme)
    ~~~~~
```

NameError: name 'somme' is not defined. Did you mean: 'somme'?

Deux-points oublié : `SyntaxError: expected ':'` ; bloc non décalé : `IndentationError`.

Lire un message d'erreur (2)

TypeError : types incompatibles

```
1 n = 5
2 print("n = " + n)
```

```
Traceback (most recent call last):
  File "script.py", line 2, in <module>
    print("n = " + n)
                ~~~~~^~~
```

TypeError: can only concatenate str (not "int") to str

On écrit `"n = " + str(n)`, ou mieux `print(f"n = {n}")`. Même erreur après un `input` non converti.

ZeroDivisionError

```
1 a = 10
2 b = 0
3 print(a / b)
```

```
Traceback (most recent call last):
  File "script.py", line 3, in <module>
    print(a / b)
          ~~~~
```

ZeroDivisionError: division by zero

Même erreur pour `//` et `%`, typiquement quand un diviseur `d` part de `0` au lieu de `1`. Le libellé exact dépend de la version de Python ; une erreur signalée ligne `k` a parfois sa cause ligne `k - 1`.

Écrire et tester un petit programme pas à pas

Méthode : compter les diviseurs positifs de $n \geq 1$

1. **Entrée et sortie** : n fixé au début du script ; on affiche le nombre de diviseurs.
2. **Un petit cas à la main** : $n = 12$, diviseurs 1, 2, 3, 4, 6, 12 ; réponse attendue 6.
3. **Boucle et accumulateur** : `for d in range(1, n + 1)`, `compteur` augmenté si `n % d == 0`.
4. **Exécuter sur le cas test**, avec un affichage provisoire (ligne 5).
5. **Cas limites** : $n = 1$ donne 1 ; $n = 0$ donne 0 sans erreur (`range(1, 1)` est vide).
6. **Retirer la ligne 5**, puis exécuter : pour $n = 2025$, le script affiche 15.

```
1 n = 12
2 compteur = 0
3 for d in range(1, n + 1) :
4     if n % d == 0 :
5         print(d)
6         compteur = compteur + 1
7 print("nombre de diviseurs :", compteur)
```

Sortie :

```
1
2
3
4
6
12
nombre de diviseurs : 6
```

Les erreurs classiques

- Confondre `=` (affectation) et `==` (test d'égalité).
- Oublier que `input` renvoie une chaîne : `"17" + 10` est une `TypeError`.
- Écrire `-2 ** 2` en pensant $(-2)^2$: la puissance passe avant le signe.
- Tester l'égalité de deux flottants avec `==` : `0.1 + 0.2 == 0.3` vaut `False`.
- Utiliser `/` dans un calcul entier : `6 / 2` vaut `3.0`, un flottant.
- Oublier que la borne de droite de `range` est exclue : `[1, n]` s'écrit `range(1, n+1)`.
- Oublier de faire évoluer la variable de la condition d'un `while` : boucle infinie.
- Lire la valeur de sortie d'un `while` sans tenir compte du décalage d'un tour.
- Mal indenter : l'indentation (quatre espaces) fait partie de la syntaxe.

Ce qu'il faut retenir

Valeurs et opérations

- Affectation `nom = expression` : la droite est évaluée d'abord. Types `int`, `float`, `bool`, `str`.
- Division euclidienne : quotient `a // b`, reste `a % b`; `b` divise `a` $\iff a \% b == 0$.
- Affichage lisible : `print(f"n = {n}")`.
- `and`, `or`, `not` traduisent *et*, *ou*, *non*; $P \implies Q$ s'écrit `(not P) or Q`.

Contrôle du programme

- `if` / `elif` / `else` : un seul bloc exécuté, le premier dont la condition est vraie.
- `for` quand le **nombre de tours est connu** ($\llbracket 1, n \rrbracket$, calcul de u_n); `while` quand il ne l'est pas (seuil, premier diviseur). Tout `for` se réécrit en `while`, pas l'inverse.
- Accumulateurs : somme initialisée à 0, produit à 1, compteur à 0, maximum au premier terme.
- Une erreur se lit par sa **dernière ligne**; un programme s'écrit par petites touches, testé sur un cas fait à la main.

Fin du chapitre 32

Place au chapitre 33 : Fonctions, récursivité et listes