MON APPRENTISSAGE — SERIES PYTHON

Python

Du zéro à l'expert

Chapitre 4 — Les Entrées et Sorties

Faire parler Python — et lui faire écouter

Auteur :
KETOTSA AMÉVI CLAUDE

Date de publication :
2 mars 2026

Publié dans le cadre de mon parcours d'apprentissage Python

Table des matières

Chapitre 4 — Les Entrées et Sorties

À retenir

Un programme qui ne communique pas ne sert à rien. Dans ce chapitre, j'apprends comment Python **affiche des informations** à l'utilisateur avec `print()`, et comment il **reçoit des données** avec `input()`. C'est ici que les programmes commencent à véritablement *vivre*.

La fonction print() — Afficher des données

`print()` est la fonction la plus utilisée en Python. Elle affiche n'importe quelle valeur dans le terminal.

print() — Les bases

```
# Afficher du texte simple
print("Bonjour tout le monde !")

# Afficher plusieurs valeurs separees par une virgule
print("Nom :", "Alice", "| Age :", 25)
#     Nom : Alice | Age : 25

# Afficher une variable
prenom = "Bob"
age = 30
print(prenom)
print(age)

# Afficher plusieurs variables ensemble
print(prenom, age)
#     Bob 30
```

Les paramètres avancés de print()

`print()` possède des paramètres cachés qui le rendent très puissant.

Paramètres sep et end

```
# sep     definit le separateur entre les valeurs
# (par defaut : un espace)
print("Paris", "Lyon", "Marseille", sep=" | ")
#     Paris | Lyon | Marseille

print("02", "03", "2026", sep="-")
```

```
#     02-03-2026

print("A", "B", "C", sep="")
#     ABC

# end     definit ce qui s'affiche a la fin
# (par defaut : saut de ligne \n)
print("Chargement", end="")
print("...")
#     Chargement...

print("Etape 1", end="     ")
print("Etape 2", end="     ")
print("Etape 3")
#     Etape 1     Etape 2     Etape 3
```

Ce qui m'a résisté

Le paramètre `end` m'a surpris au début. Je pensais que `print()` ajoutait toujours un retour à la ligne — c'est son comportement **par défaut**, mais on peut totalement le contrôler. Comprendre que `print("texte")` est en réalité `print("texte", end="\n")` a changé ma façon de voir la fonction.

Le formatage des chaînes

Python offre trois façons de formater du texte. La méthode moderne et recommandée est le **f-string**.

Les 3 méthodes de formatage

```
prenom = "Alice"
age = 25
taille = 1.68

# Methode 1     Concatenation (ancienne, deconseillee)
print("Je m'appelle " + prenom + " et j'ai " + str(age) + " ans
    .")

# Methode 2     format() (intermediaire)
print("Je m'appelle {} et j'ai {} ans.".format(prenom, age))

# Methode 3     f-string (moderne, recommandee    )
print(f"Je m'appelle {prenom} et j'ai {age} ans.")
#     Je m'appelle Alice et j'ai 25 ans.
```

f-strings — Puissance et précision

```python
prix = 19.9999
pi = 3.14159265

# Arrondir les decimales dans un f-string
print(f"Prix : {prix:.2f}     ")      #      Prix : 20.00
print(f"Pi     {pi:.4f}")             #      Pi     3.1416

# Aligner du texte sur une largeur fixe
print(f"{'Produit':<15} {'Prix':>8}")
print(f"{'Caf  ':<15} {2.50:>8.2f}")
print(f"{'Croissant':<15} {1.20:>8.2f}")
#
# Produit               Prix
# Cafe                  2.50
# Croissant             1.20

# Expressions dans les f-strings
a = 10
b = 3
print(f"{a} x {b} = {a * b}")    #      10 x 3 = 30
print(f"Majeur : {a >= 18}")     #      Majeur : False
```

Ce qui m'a résisté

La syntaxe `:.2f` dans les f-strings m'a demandé du temps. Le `.2f` signifie : *"affiche ce nombre flottant avec exactement 2 décimales"*. Le `<` et `>` pour l'alignement m'ont aussi surpris — mais une fois compris, c'est redoutablement pratique pour créer des tableaux dans le terminal.

Les caractères spéciaux dans les chaînes

Python utilise des **séquences d'échappement** pour représenter des caractères spéciaux.

`\n`	Saut de ligne
`\t`	Tabulation horizontale
`\r`	Retour chariot
`\\`	Antislash littéral
`\"`	Guillemet double dans une chaîne
`\'`	Guillemet simple dans une chaîne

Caractères spéciaux en action

```
# Saut de ligne \n
print("Ligne 1\nLigne 2\nLigne 3")

# Tabulation \t
print("Nom\tAge\tVille")
print("Alice\t25\tParis")
print("Bob\t30\tLyon")

# Chemin de fichier Windows      attention au \
#      Mauvais :  print("C:\nouveau\fichier.txt")
#      Solution 1 : doubler les antislashs
print("C:\\nouveau\\fichier.txt")
#      Solution 2 : raw string avec r""
print(r"C:\nouveau\fichier.txt")
```

La fonction input() — Recevoir des données

`input()` permet au programme de **demander une information** à l'utilisateur. Il s'arrête et attend que l'utilisateur tape quelque chose et appuie sur Entrée.

input() — Les bases

```
# Demander une information
prenom = input("Quel est ton prenom ? ")
print(f"Bonjour, {prenom} !")

# input() retourne TOUJOURS une chaine (str)
age_texte = input("Quel est ton age ? ")
print(type(age_texte))    # <class 'str'>

# Convertir pour faire des calculs
age = int(input("Quel est ton age ? "))
annee_naissance = 2026 - age
print(f"Tu es ne(e) en {annee_naissance}.")
```

Programme interactif complet

```
# Mini programme de presentation
print("=" * 40)
print("   GENERATEUR DE PRESENTATION")
print("=" * 40)

prenom = input("\nTon prenom : ")
age    = int(input("Ton age : "))
ville  = input("Ta ville : ")
metier = input("Ton metier : ")
```

```
print("\n" + "=" * 40)
print(f"  Bonjour, je m'appelle {prenom}.")
print(f"  J'ai {age} ans et j'habite a {ville}.")
print(f"  Je travaille comme {metier}.")
print(f"  Dans {65 - age} ans, je serai a la retraite !")
print("=" * 40)
```

Ce qui m'a résisté

La subtilité qui m'a le plus piégé : `input()` retourne **toujours** une chaîne de caractères — même si l'utilisateur tape 25. J'ai eu une erreur en essayant d'additionner directement le résultat d'un `input()` avec un nombre. Python ne convertit pas automatiquement. Il faut **toujours** encadrer avec `int()` ou `float()` si on veut faire des calculs. Cette règle semble simple, mais elle m'a coûté plusieurs minutes de débogage avant de comprendre.

Gérer les erreurs de saisie

Que se passe-t-il si l'utilisateur tape du texte alors qu'on attend un nombre ?

Erreur classique et protection

```
#     Si l'utilisateur tape "abc" au lieu d'un nombre
# int(input(...)) va lever une erreur : ValueError

#     Protection avec try/except (apercu     chapitre 19)
try:
    age = int(input("Ton age : "))
    print(f"Dans 10 ans, tu auras {age + 10} ans.")
except ValueError:
    print("Erreur : veuillez entrer un nombre entier.")

#     Validation simple avec isdigit()
saisie = input("Entrez un nombre : ")
if saisie.isdigit():
    nombre = int(saisie)
    print(f"Votre nombre au carre : {nombre ** 2}")
else:
    print("Ce n'est pas un nombre valide.")
```

Note importante

La gestion complète des erreurs sera couverte au **Chapitre 19**. Mais il est important de savoir dès maintenant que `int(input())` peut *planter* si l'utilisateur ne coopère pas. Un bon programme anticipe toujours les erreurs humaines.

Ce qui m'a le plus surpris dans ce chapitre

Ce qui m'a résisté

Trois subtilités qui m'ont marqué :

1. La multiplication de chaînes. En découvrant `print("=" * 40)`, j'ai réalisé qu'on pouvait multiplier une chaîne par un nombre en Python. C'est élégant et inattendu — `"=" * 40` produit une ligne de 40 signes égal. Aucun autre langage que je connaisse ne fait ça aussi naturellement.

2. Les f-strings sont des expressions, pas juste du texte. On peut écrire `f"{2 + 2}"` et obtenir `4`. On peut appeler des fonctions, faire des calculs, tout ça à l'intérieur des accolades. Ce n'est pas juste du formatage — c'est du Python à part entière, intégré dans du texte.

3. Le `r""` pour les raw strings. Écrire un chemin Windows comme `C:\Users\fichier` provoque des erreurs à cause des antislashs. La solution `r"C:\Users\fichier"` désactive l'interprétation des séquences d'échappement. Un détail technique qui peut faire perdre beaucoup de temps si on ne le connaît pas.

Résumé du Chapitre 4

Ce que j'ai appris dans ce chapitre :

- `print()` affiche n'importe quelle valeur — avec `sep` et `end` pour contrôler la mise en forme.
- Les **f-strings** sont la méthode moderne pour formater du texte — puissantes et lisibles.
- `:.2f` dans un f-string contrôle la précision des décimales.
- Les **séquences d'échappement** : `\n`, `\t`, `r""` pour les raw strings.
- `input()` reçoit toujours une `str` — toujours convertir avec `int()` ou `float()`.
- Anticiper les erreurs de saisie avec `isdigit()` ou `try/except`.
- `"=" * 40` — multiplier une chaîne par un entier est une des élégances uniques de Python.

"Un programme qui ne communique pas avec l'humain n'est qu'une boîte noire. L'entrée et la sortie, c'est le dialogue."