



Programmation Python

Séance 4 — Fonctions, modules et robustesse

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/programmation-python/chapitres/04-fonctions-modules-robustesse.html>

Version du 11 octobre 2026



Version en ligne

Programmation Python

— Séance 4

Fonctions, modules et robustesse

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

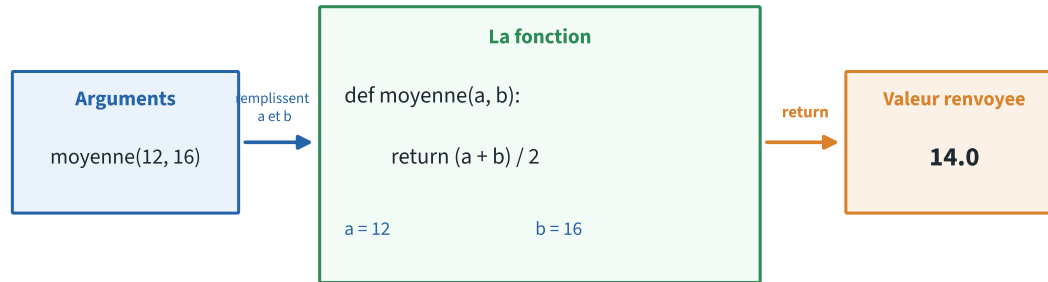
Contenu

À mesure qu'un programme grandit, on répète du code et on rencontre des erreurs. Cette séance apprend à **organiser** (des fonctions réutilisables), à **réutiliser** (des modules tout prêts), à **conserver** (des fichiers) et à **fiabiliser** (gérer les erreurs).

- **Fonctions** : `def`, `return`, paramètres, valeurs par défaut, docstrings, portée.
- **Modules** : `import` pour réutiliser la bibliothèque standard.
- **Fichiers** : lire et écrire des données sur le disque.
- **Exceptions** : `try` / `except` pour ne pas planter.

L'entraînement se fait dans le **notebook**, avec une cellule « À vous » après chaque notion.

Anatomie d'un appel de fonction



Les arguments (12, 16) remplissent les paramètres (a, b) ; return renvoie 14.0, reutilisable dans un calcul.

Les arguments fournis à l'appel remplissent les paramètres ; le corps s'exécute ;
`return` renvoie une valeur.

Les fonctions

L'idée : un bloc de code réutilisable

L'idée en une phrase

Une **fonction** est un bloc nommé qu'on définit une fois avec `def`, puis qu'on **appelle** autant de fois que nécessaire. `return` renvoie un résultat que l'on peut réutiliser.

Le bénéfice est double : on **évite de répéter** le même code (une correction se fait à un seul endroit), et on **donne un nom** à une opération, ce qui rend le programme lisible. Le corps de la fonction est un bloc indenté après les deux points.

Définir et appeler

```
def moyenne(a, b):  
    return (a + b) / 2  
  
print(moyenne(12, 16))  
print(moyenne(9, 18))
```

RÉSULTAT →

```
14.0  
13.5
```

a et **b** sont les **paramètres** ; 12 et 16 les **arguments** de l'appel. La même fonction sert pour n'importe quelles valeurs.

Plusieurs paramètres

```
def message(prenom, ville):  
    return f"{prenom} habite a {ville}."  
  
print(message("Fatou", "Dakar"))
```

RÉSULTAT →

Fatou habite a Dakar.

Les arguments sont associés aux paramètres **dans l'ordre** : `prenom` reçoit « Fatou », `ville` reçoit « Dakar ».

Valeurs par défaut

Un paramètre peut avoir une valeur par défaut

Si l'argument n'est pas fourni, la valeur par défaut est utilisée.

```
def saluer(nom, politesse="Bonjour"):  
    return f"{politesse} {nom}"
```

```
print(saluer("Awa"))  
print(saluer("Awa", "Bonsoir"))
```

RÉSULTAT →

```
Bonjour Awa  
Bonsoir Awa
```

Documenter : la docstring

```
def carre(x):  
    "Renvoie le carre de x."  
    return x * x
```

```
print(carre(5))  
print(carre.__doc__)
```

RÉSULTAT →

```
25  
Renvoie le carre de x.
```

La **docstring** est une courte phrase placée juste sous le `def`. Elle explique ce que fait la fonction, sans qu'on ait à lire son code.

Piège — la portée d'une variable

Une variable créée dans une fonction est locale

`resultat` n'existe que dans `calcul`. Y accéder à l'extérieur échoue.

```
def calcul():  
    resultat = 5 * 3
```

```
calcul()  
print(resultat)
```

```
NameError: name 'resultat' is not defined
```

Pour récupérer une valeur calculée dans une fonction, il faut la **renvoyer** avec `return`.

return n'est pas print

```
def somme(a, b):  
    return a + b  
  
def affiche_somme(a, b):  
    print(a + b)  
  
x = somme(10, 5) + 100  
print("x =", x)  
y = affiche_somme(10, 5)  
print("y =", y)
```

RÉSULTAT →

```
x = 115  
15  
y = None
```

return renvoie une valeur (réutilisable dans un calcul) ; une fonction qui se contente d'afficher renvoie **None**.

Les modules

L'idée : réutiliser du code tout prêt

L'idée en une phrase

Un **module** est une bibliothèque de code déjà écrit, qu'on **importe** pour l'utiliser. Python en fournit une riche collection en standard — inutile de tout réécrire.

Le module `math` donne les outils mathématiques, `random` le hasard, `datetime` les dates, etc. On importe une fois, en haut du fichier, puis on utilise. C'est un premier pas vers l'écosystème `numpy` / `pandas` qu'on découvrira à partir de la Séance 6.

Importer un module : `import`

```
import math  
  
print(math.sqrt(16))  
print(math.pi)
```

RÉSULTAT →

```
4.0  
3.141592653589793
```

Après `import math`, on accède à ses outils en les préfixant par `math.` — ce qui indique clairement d'où ils viennent.

Importer un seul outil : `from ... import`

```
from math import sqrt
```

```
print(sqrt(25))
```

RÉSULTAT →

5.0

On importe **seulement** ce dont on a besoin ; `sqrt` s'utilise alors directement, sans préfixe. Pratique pour quelques fonctions bien identifiées.

Les fichiers

L'idée : conserver des données sur le disque

L'idée en une phrase

Les variables disparaissent à la fin du programme. Un **fichier** permet de **sauvegarder** des données et de les retrouver plus tard. On l'ouvre avec `open()` dans un bloc `with`, qui le **referme automatiquement**.

Le **mode** précise l'intention : `"w"` écrit (et écrase le contenu existant), `"a"` ajoute à la fin, `"r"` lit (c'est le mode par défaut). Un fichier texte est une simple suite de lignes, séparées par `\n`.

Écrire dans un fichier

```
with open("villes.txt", "w") as f:  
    f.write("Dakar\n")  
    f.write("Louga\n")  
    f.write("Mbour\n")  
  
print("fichier écrit")
```

RÉSULTAT →

fichier écrit

Le bloc `with` ouvre `villes.txt` en écriture, y écrit trois lignes, puis le referme tout seul en sortant du bloc. Le `\n` marque la fin de chaque ligne.

Lire un fichier

Tout le contenu d'un coup, avec `.read()` :

```
with open("villes.txt") as f:
    contenu = f.read()
print(contenu)
```

Ou **ligne par ligne**, avec une boucle `for` (le plus courant) :

```
with open("villes.txt") as f:
    for ligne in f:
        print(ligne.strip())
```

RÉSULTAT →

Dakar
Louga
Mbour

Piège — le mode d'ouverture

"w" écrase tout

Ouvrir un fichier existant en mode "w" **efface son contenu** avant d'écrire. Pour **compléter** un fichier sans le vider, on utilise le mode "a" (append). Pour seulement le lire, "r" (ou rien, c'est le défaut) — et écrire dans un fichier ouvert en lecture lève une erreur.

Toujours privilégier le bloc `with` : il garantit que le fichier est refermé, même si une erreur survient au milieu.

Les exceptions

L'idée : rattraper une erreur au lieu de planter

L'idée en une phrase

Certaines opérations **échouent** à l'exécution. Sans précaution, le programme s'arrête net :

```
nombre = int("abc")
```

```
ValueError: invalid literal for int() with base 10: 'abc'
```

On **essaie** le code risqué dans un bloc **try**, et on **rattrape** l'erreur dans un bloc **except** — le programme continue au lieu de s'interrompre.

try / except

```
try:
    nombre = int("abc")
    print(nombre)
except ValueError:
    print("Ce n'est pas un nombre valide.")
```

RÉSULTAT →

Ce n'est pas un nombre valide.

Le code du `try` s'exécute ; dès qu'une `ValueError` survient, on saute au `except`, qui affiche un message clair. Le programme n'a pas planté.

Une fonction robuste

Renvoyer une valeur de repli plutôt que planter

On précise **quelle** erreur on rattrape (`ZeroDivisionError`).

```
def division_sure(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError:  
        return "impossible"  
  
print(division_sure(10, 2))  
print(division_sure(10, 0))
```

RÉSULTAT →

```
5.0  
impossible
```

Piège — ne pas tout masquer

Rattraper l'erreur attendue, pas toutes

On indique l'erreur précise qu'on sait gérer (`ValueError`, `ZeroDivisionError`, `FileNotFoundError`...). Un `except sans type`, qui rattrape *tout*, cache aussi les erreurs qu'on n'avait pas prévues (une faute de frappe dans un nom, par exemple) et rend les bugs très difficiles à trouver.

Règle : rattraper le **moins** possible — seulement ce que l'on sait traiter — et laisser remonter le reste.

Synthèse

À retenir — Séance 4

L'essentiel

Fonction : `def nom(parametres): + return` ; appelée avec des **arguments** ; peut avoir des **valeurs par défaut** et une **docstring**.

Portée : une variable définie dans une fonction est **locale** ; pour en sortir une valeur, on la `return`. `return` $\neq$ `print`.

Modules : `import math` puis `math.sqrt`, ou `from math import sqrt`.

Fichiers : `with open("f.txt", "w") as f` : pour écrire, `"r"` pour lire ; `.read()` ou boucle `for` ligne par ligne ; `"w"` écrase, `"a"` ajoute.

Exceptions : `try` : le code risqué, `except` l'erreur **précise** ; ne pas tout masquer.

Pièges fréquents

- Utiliser dehors une variable **locale** à une fonction : `NameError` — la `return`.
- Attendre une valeur d'une fonction qui fait `print` au lieu de `return` (elle renvoie `None`).
- Oublier les deux points `:` ou l'indentation du corps de la fonction.
- Ouvrir en mode `"w"` un fichier qu'on voulait compléter : son contenu est **effacé**.
- Rattraper **toutes** les erreurs avec un `except` nu : les vrais bugs deviennent invisibles.

Prochaine séance

Séance 5 — Programmation orientée objet

On sait écrire des fonctions. La Séance 5 introduit les **objets** : regrouper des données et les fonctions qui les manipulent dans une même **classe** (`class`, `__init__`, `self`, méthodes), puis **spécialiser** une classe par **héritage** et **polymorphisme**.

À faire d'ici là : refaire les cellules « À vous » du notebook de la séance.

Références

- Severance, C. — *Python for Everybody*, 2016.
- Downey, A. — *Think Python*, 2e éd., O'Reilly, 2015.
- Matthes, E. — *Python Crash Course*, 3e éd., No Starch Press, 2023.
- Python Software Foundation — *The Python Tutorial* (Functions, Modules, Errors and Exceptions), 2024.

Ressources de la séance

- **Notebook** · [TP 4 — Fonctions, modules et robustesse](#) (15 Ko)