



Programmation Python

Séance 0 — Mise en place et outils du praticien

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/00-mise-en-place-et-outils.html>

Version du 11 octobre 2026



Version en ligne

Programmation Python

— Séance 0

Mise en place et outils du praticien

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

Avant d'écrire la moindre ligne de Python, il faut **installer l'atelier** et savoir s'en servir. Cette séance ne contient aucun formalisme : on met en place les outils, et on apprend les gestes qui reviendront à chaque séance. Le but n'est pas de tout retenir, mais d'avoir un poste de travail qui fonctionne et de ne plus paniquer devant un message d'erreur.

- **Le terminal** : donner des ordres à la machine en quelques commandes.
- **Anaconda et les environnements** : installer Python dans une bulle isolée.
- **Jupyter** : le cahier interactif où l'on écrit et exécute du code.
- **Lire une erreur** : comprendre un *traceback* au lieu de le craindre.
- **Git et GitHub** : sauvegarder et partager son travail (découverte).

Le détail d'installation, étape par étape, est dans la **fiche pratique** fournie avec la séance.

Pourquoi des outils ?

L'idée : programmer, c'est tenir un atelier

Analogie

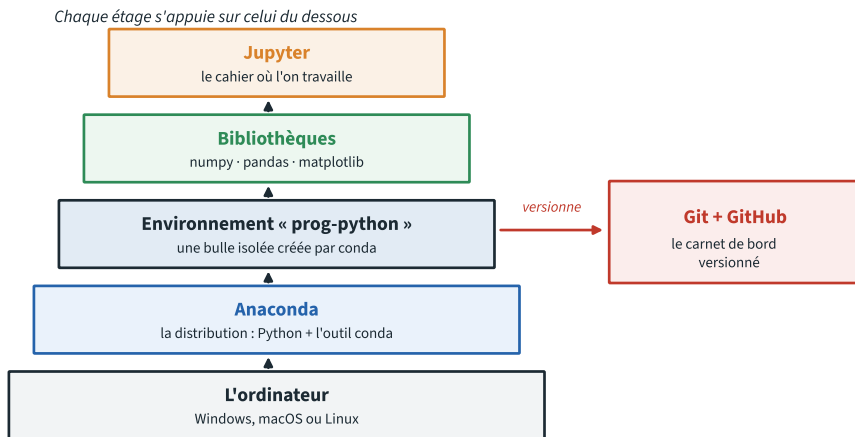
Un menuisier ne se résume pas à son marteau : il a un **établi**, des **outils rangés**, et un **carnet** où il note ce qu'il fait. Programmer, c'est pareil. Le code n'est qu'une partie ; autour, il y a un atelier qu'il faut monter une fois pour toutes.

Quatre outils composent cet atelier, et chacun a un rôle précis :

- le **terminal**, pour donner des ordres directs à la machine ;
- **Anaconda**, qui installe Python et range les bibliothèques dans des **environnements** ;
- **Jupyter**, le cahier où l'on travaille réellement ;
- **Git** (avec **GitHub**), le carnet de bord qui garde la trace de tout.

On les installe d'abord, on les comprend ensuite. La séance suit cet ordre.

L'atelier du praticien



Chaque outil s'appuie sur le précédent ; Git veille à côté et conserve l'historique du travail.

Le terminal

L'idée : parler à la machine en ligne de commande

L'idée en une phrase

Le **terminal** est une fenêtre où l'on tape des **commandes** que la machine exécute aussitôt. Pas de boutons, pas de souris : on écrit l'ordre, on appuie sur Entrée, on lit la réponse.

C'est austère, mais c'est l'outil le plus direct et le plus universel : installer Python, créer un environnement, lancer Jupyter, utiliser Git — tout passe par là. On l'ouvre via *Terminal* (macOS, Linux) ou *Anaconda Prompt* (Windows). Le signe \$ au début d'une ligne signale « ici, vous tapez une commande » ; il ne se tape pas.

Les commandes vitales

Se repérer et créer des dossiers

Quatre commandes suffisent pour commencer : savoir où l'on est, voir ce qu'il y a, se déplacer, créer un dossier.

```
$ pwd                                # ou suis-je ? (Print Working Directory)
/home/fatou
$ ls                                  # lister le contenu du dossier
Bureau  Documents  Telechargements
$ mkdir projet-python                # creer un dossier
$ cd projet-python                   # entrer dedans
$ pwd
/home/fatou/projet-python
```

`cd ..` remonte d'un cran ; `cd` seul ramène au dossier personnel.

Piège — les chemins

Chemin relatif ou absolu

Un **chemin absolu** part de la racine et ne change jamais : `/home/fatou/projet-python`.

Un **chemin relatif** part de l'endroit où l'on se trouve : `projet-python` ne marche que si l'on est déjà dans `/home/fatou`. Une commande qui « ne trouve pas le fichier » vient presque toujours de là : on n'est pas dans le bon dossier (`pwd` pour vérifier).

Second piège : les **espaces** dans les noms. `cd Mon projet` échoue, car le terminal lit deux mots. On évite les espaces (`mon-projet`) ou on protège par des guillemets : `cd "Mon projet"`.

Anaconda et les environnements

L'idée : une boîte à outils isolée par projet

L'idée en une phrase

Un **environnement** est une installation de Python séparée, avec ses propres bibliothèques, réservée à un projet. **Anaconda** fournit Python et l'outil **conda** qui crée et gère ces environnements.

Analogie

Une **trousse par matière** : les stylos de mathématiques ne se mélangent pas avec ceux de dessin. De même, un projet qui a besoin d'une ancienne version d'une bibliothèque ne doit pas casser un autre projet. Chaque environnement est une bulle : ce qu'on y installe n'affecte que lui.

Créer et activer un environnement

```
$ conda create -n prog-python python=3.12
Collecting package metadata: done
Solving environment: done
Proceed ([y]/n)? y
Preparing transaction: done
#
# To activate this environment, use
#     $ conda activate prog-python

$ conda activate prog-python
(prog-python) $      # le nom de la bulle apparait a gauche
```

-n prog-python nomme l'environnement ; python=3.12 fixe la version. Une fois *activé*, tout ce qu'on installe va dans cette bulle.

Installer des bibliothèques

Ajouter les outils de la science des données

Dans l'environnement activé, on installe les bibliothèques du cours en une commande.

```
(prog-python) $ pip install numpy pandas matplotlib jupyter
...
Successfully installed jupyter matplotlib numpy pandas

(prog-python) $ conda list          # verifier ce qui est installe
```

L'erreur numéro un

`ModuleNotFoundError: No module named 'pandas'` signifie presque toujours :
l'environnement n'est pas activé, ou la bibliothèque a été installée dans la mauvaise bulle.
Réflexe : vérifier le `(prog-python)` à gauche, puis réactiver.

Une alternative : venv

Pour information

Sans Anaconda, Python possède son propre outil d'environnements, `venv` : `python -m venv mon-env` crée la bulle, puis on l'active. C'est plus léger mais moins complet que `conda` (qui gère aussi des outils non-Python). Pour ce cours, on reste sur `conda`.

À retenir simplement : **toujours travailler dans un environnement activé**, jamais dans le Python « du système ». C'est ce qui rend un projet reproductible : on peut recréer exactement la même bulle sur une autre machine.

Jupyter, le carnet de laboratoire

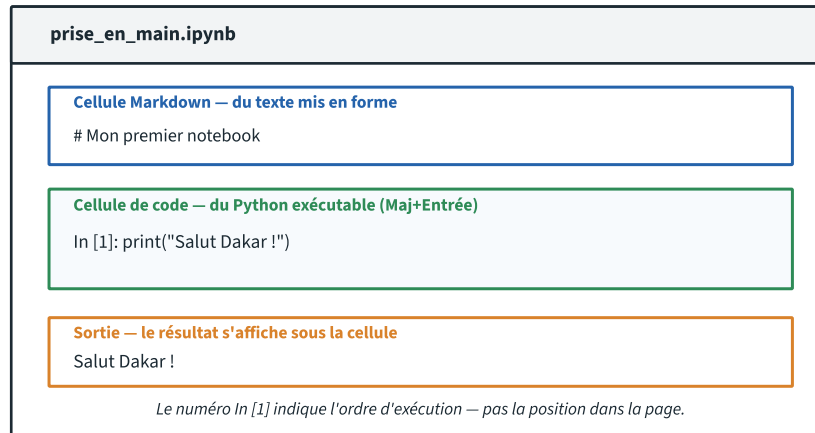
L'idée : un cahier qui mêle texte, code et résultats

L'idée en une phrase

Un **notebook** Jupyter est un document fait de **cellules** : des cellules de *texte* (Markdown) pour expliquer, des cellules de *code* (Python) qu'on exécute une à une, et la *sortie* qui s'affiche juste en dessous.

C'est l'outil du praticien des données : on essaie une idée, on voit le résultat immédiatement, on ajuste. On peut tout raconter dans le même document — d'où son nom de cahier de laboratoire. Tous les TP du cours sont des notebooks.

Anatomie d'un notebook



Trois types de contenu dans un même document : texte, code exécutable, et la sortie produite.

Lancer Jupyter et exécuter une cellule

```
(prog-python) $ jupyter notebook  
[I] Jupyter Notebook is running at:  
[I] http://localhost:8888/tree
```

Le navigateur s'ouvre. On crée un notebook, on tape une ligne dans une cellule de code, puis **Maj+Entrée** pour l'exécuter :

```
print("Salut Dakar !")
```

RÉSULTAT →

```
Salut Dakar !
```

Le compteur passe de In `[\,]` à In `[1]` : la cellule a été exécutée, et c'était la première.

Piège — l'ordre d'exécution

Le notebook garde une mémoire cachée

Les cellules ne s'exécutent pas forcément du haut vers le bas : elles s'exécutent **dans l'ordre où vous appuyez sur Maj+Entrée**. Le numéro `In [n]` donne cet ordre réel. Si vous revenez modifier une cellule du haut sans ré-exécuter celles du bas, l'affichage et l'état réel ne correspondent plus.

Symptôme classique : une variable « existe » alors qu'on a effacé la cellule qui la définissait — elle vit encore en mémoire. Réflexe en cas de doute : menu *Kernel* → *Restart & Run All* pour tout ré-exécuter proprement, du début.

Lire une erreur sans paniquer

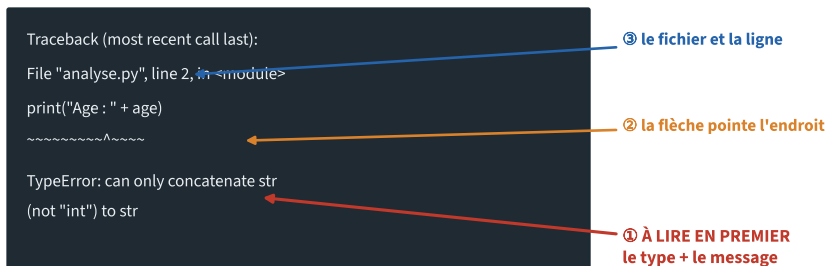
L'idée : une erreur est un message, pas un échec

L'idée en une phrase

Quand Python rencontre un problème, il s'arrête et affiche un **traceback** : un message qui indique *où* ça casse et *pourquoi*. Ce n'est pas une punition, c'est une information précise. La règle : le **lire de bas en haut**.

La dernière ligne donne le **type** d'erreur et sa cause — c'est elle qu'on lit en premier. Au-dessus, le fichier et le numéro de ligne situent l'endroit. Une flèche (^ ou ^^) pointe le caractère fautif. Avec un peu d'habitude, les mêmes erreurs reviennent : on les reconnaît d'un coup d'œil.

Anatomie d'un traceback



Un traceback se lit de BAS en HAUT : la dernière ligne dit le pourquoi.

La dernière ligne (le type et le message) est la plus utile ; on remonte ensuite vers l'endroit fautif.

Erreur 1 — SyntaxError

Python n'arrive pas à *lire* le code : il est mal écrit.

```
print("Bonjour Dakar)
```

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

```
SyntaxError: unterminated string literal (detected at line 1)
```

unterminated string literal = un guillemet n'a pas été fermé. Correction : `print("Bonjour Dakar")`. Ce type d'erreur empêche le programme de démarrer.

Erreur 2 — NameError

On utilise un nom que Python ne connaît pas (souvent une faute de frappe).

```
moyenne = 14.5  
print(moyene)
```

```
Traceback (most recent call last):  
  File "analyse.py", line 2, in <module>  
    print(moyene)  
    ^^^^^^^  
NameError: name 'moyene' is not defined. Did you mean: 'moyenne'?
```

Python va jusqu'à **suggérer** le bon nom. Correction : `print(moyenne)`.

Erreur 3 — IndentationError

L'indentation (le décalage en début de ligne) est obligatoire en Python.

```
for i in range(3):  
print(i)
```

```
File "analyse.py", line 2  
    print(i)  
    ^
```

```
IndentationError: expected an indented block after 'for' statement on line 1
```

Le corps d'une boucle doit être **décalé** (4 espaces). Correction : indenter `print(i)`. On reverra l'indentation en Séance 3.

Erreur 4 — TypeError

On combine des types incompatibles : ici du texte et un entier.

```
age = 23
print("Age : " + age)
```

```
Traceback (most recent call last):
  File "analyse.py", line 2, in <module>
    print("Age : " + age)
    ~~~~~^~~~~
TypeError: can only concatenate str (not "int") to str
```

On ne « colle » pas un nombre à du texte. Correction propre, avec une f-string (Séance 1): `print(f"Age : {age}")`.

Git et GitHub (découverte)

L'idée : une machine à remonter le temps pour le code

L'idée en une phrase

Git enregistre des **photos** successives de votre projet (les *commits*). On peut revenir à n'importe quelle photo, comparer, ne jamais rien perdre. **GitHub** est un site qui héberge une copie en ligne de ces photos, pour les sauvegarder et les partager.

Analogie

Comme des **sauvegardes nommées** d'un document : « version avec l'introduction », « version corrigée ». Sauf qu'ici chaque sauvegarde est datée, accompagnée d'un message, et qu'on peut revenir à l'une d'elles à tout moment.

Le cycle de Git



Chaque commit est une sauvegarde datée et nommée ; push l'envoie sur GitHub.

De vos fichiers à GitHub : `add` prépare, `commit` enregistre une photo, `push` l'envoie en ligne.

Le cycle en pratique

Premier dépôt, premier commit

On initialise un dépôt dans le dossier du projet, on prépare un fichier, on enregistre une photo.

```
$ git init
Initialized empty Git repository in /home/fatou/projet/.git/
$ git add analyse.py
$ git commit -m "Premier commit : squelette de l'analyse"
[main (root-commit) f07be7e] Premier commit : squelette de l'analyse
1 file changed, 2 insertions(+)
create mode 100644 analyse.py
```

f07be7e est l'identifiant unique de cette photo ; le message décrit ce qu'elle contient.

Envoyer sur GitHub

Après avoir créé un dépôt vide sur GitHub, on le relie au projet local et on envoie :

```
$ git remote add origin https://github.com/fatou/projet.git
$ git push -u origin main
To https://github.com/fatou/projet.git
 * [new branch]      main -> main
```

Un fichier `.gitignore` indique ce qu'il ne faut **pas** versionner :

```
# .gitignore
__pycache__/
*.csv
.env
```

Pièges de Git, et ce qui vient après

Trois réflexes

1. Un commit sans `add` préalable n'enregistre rien : `add` puis `commit`. **2.** Un message de commit doit décrire le *quoi* (« ajoute le calcul de la moyenne »), pas « modif ». **3.** Ne jamais pousser de **données** volumineuses ni de **mots de passe** : c'est le rôle du `.gitignore`.

Vu plus loin

Git permet aussi de travailler à plusieurs grâce aux *branches* et au *merge*. Ces notions ne sont pas nécessaires ici : à ce stade, le cycle `add` → `commit` → `push` suffit largement.

Synthèse

À retenir — Séance 0

L'essentiel

Le terminal exécute des commandes : `pwd`, `ls`, `cd`, `mkdir`.

Anaconda installe Python ; `conda` crée des **environnements** isolés. On travaille toujours dans un environnement **activé**.

Jupyter est le cahier : cellules de texte, de code, et sorties. Le numéro `In [n]` donne l'ordre réel d'exécution.

Un traceback se lit de bas en haut : le type et le message d'abord, puis l'endroit.

Git enregistre des photos du projet : `add` → `commit` → `push` ; **GitHub** en garde une copie en ligne.

Pièges fréquents

- « **Fichier introuvable** » au terminal : on n'est pas dans le bon dossier (`pwd`), ou un espace traîne dans un nom.
- **ModuleNotFoundError** : environnement non activé, ou bibliothèque installée dans la mauvaise bulle.
- **Notebook incohérent** : cellules exécutées dans le désordre ; *Restart & Run All*.
- **Lire le haut du traceback** au lieu du bas : la cause est sur la **dernière** ligne.
- **commit sans add**, ou message vague : préparer puis enregistrer, avec un message clair.
- **Pousser des données ou des secrets** sur GitHub : les exclure via `.gitignore`.

Prochaine séance

Séance 1 — Valeurs, types et chaînes

L'atelier est monté : on peut écrire du Python. La Séance 1 pose les premières briques du langage — les **variables**, les **types** (entier, décimal, texte, booléen), les **opérations**, les **f-strings** pour afficher proprement, et la manipulation des **chaînes de caractères**.

À faire d'ici là : suivre la **fiche pratique** pour installer Anaconda, créer l'environnement [prog-python](#), lancer Jupyter, et déposer un premier fichier sur GitHub.

Références

- Software Carpentry — *The Unix Shell et Version Control with Git*, lessons, 2024.
- Anaconda — *Getting Started with conda*, documentation, 2024.
- Project Jupyter — *Jupyter Notebook Documentation*, 2024.
- Severance, C. — *Python for Everybody*, 2016.
- Chacon, S., Straub, B. — *Pro Git*, 2e éd., Apress, 2014.