

Module 1 : Introduction à Python

Ce module sera rapidement vu pour donner une base du langage 10-15min

✅ Objectif : Comprendre les bases de Python.

1 - Installation et configuration de l'environnement (Jupyter Notebook, VS Code)

- **Installation de python :**
 - télécharger le fichier .exe : <https://www.python.org/downloads/windows/>
 - Cocher la case ☒ "Add Python to PATH" (important pour utiliser Python en ligne de commande).
 - vérifier : ☒ Ouvre l'invite de commande (cmd) et tape : `python --version`
- **Installation de Vs Code :**
 - télécharger le fichier .exe : <https://code.visualstudio.com/>
 - Exécuter le fichier .exe et suivre l'installation en cochant "Add to PATH".
 - Installer l'extension Python dans VS Code : extensions
 - Choisir un interpréteur : `Python: Select Interpreter`
 - test : `print("Hello, Python!")`
- **Installation de Jupyter Notebook :**
 - cmd : `pip install notebook`
 - vérifier : `jupyter notebook --version`
 - lancer : `jupyter notebook`

2 - Bases de Python

📌 **Types de données (entiers, flottants, chaînes de caractères, booléens)**

Type	Description	Exemple	Fonction de conversion
int (Entier)	Nombre entier (positif ou négatif)	<code>x = 42</code>	<code>int(valeur)</code> , <code>int("10")</code> → 10
float (Flottant)	Nombre décimal	<code>x = 3.14</code>	<code>float(valeur)</code> , <code>float("3.14")</code> → 3.14
str (Chaîne de caractères)	Texte entre guillemets	<code>x = "Hello"</code>	<code>str(valeur)</code> , <code>str(42)</code> → "42"
bool (Booléen)	Valeur vraie ou fausse	<code>x = True</code>	<code>bool(valeur)</code> , <code>bool(1)</code> → True

Type	Description	Exemple	Fonction de conversion
list (Liste)	Collection ordonnée et modifiable	<code>x = [1, 2, 3]</code>	<code>list(iterable)</code>
tuple (Tuple)	Collection ordonnée et non modifiable	<code>x = (1, 2, 3)</code>	<code>tuple(iterable)</code>
dict (Dictionnaire)	Collection de paires clé-valeur	<code>x = {"nom": "Alice", "âge": 25}</code>	<code>dict(iterable)</code>
set (Ensemble)	Collection non ordonnée et sans doublons	<code>x = {1, 2, 3}</code>	<code>set(iterable)</code>
NoneType	Représente une absence de valeur	<code>x = None</code>	<i>Non applicable</i>
ndarray (Matricielle)	Matrice ou tableau multidimensionnel (via NumPy)	<code>x = np.array([[1, 2], [3, 4]])</code>	<code>np.array(liste)</code>

📌 Variables et affectations

age = 18 ← Valeur

↑

Nom de variable



source : <https://waytolearnx.com/2020/06/types-et-variables-python.html>

pour stocker en mémoire une valeur

```
mon_age = 28
nom = "Alice"
pi = 3.14
est_majeur = True
age_de_mon_pere = mon_age + 30

a, b, c = 1, 2, 3 # a = 1, b = 2, c = 3
x = y = z = 100 # x, y et z valent 100

a = 10
b = 20
```

a, b = b, a # a devient 20 et b devient 10

Les règles de nommage des variables

- ✓ Un nom de variable doit commencer par une lettre ou un underscore (_).
- ✓ Il ne peut pas contenir d'espace ni de caractères spéciaux (sauf _).
- ✓ Il ne doit pas être un mot-clé réservé de Python (if, for, def, etc.).

2variable = 5 *#! Erreur ! Ne peut pas commencer par un chiffre*

nom-utilisateur = "Bob" *# Erreur ! Pas de tiret de 6 "-"*

if = 10 *# Erreur ! "if" est un mot-clé réservé*

📌 Opérations de base et expressions

📌 Opérations arithmétiques

Opérateur	Description	Exemple	Résultat
+	Addition	5 + 3	8
-	Soustraction	10 - 4	6
*	Multiplication	6 * 2	12
/	Division (résultat flottant)	7 / 2	3.5
//	Division entière	7 // 2	3
%	Modulo (reste de la division)	10 % 3	1
**	Puissance (exponentiation)	2 ** 3	8

📌 Opérations de comparaison

Opérateur	Description	Exemple	Résultat
==	Égal à	5 == 5	True
!=	Différent de	5 != 3	True
>	Supérieur à	7 > 3	True
<	Inférieur à	4 < 2	False
>=	Supérieur ou égal à	10 >= 10	True
<=	Inférieur ou égal à	3 <= 5	True

📌 Opérations logiques

Opérateur	Description	Exemple	Résultat
and	ET logique	(5 > 3) and (4 < 6)	True

Opérateur	Description	Exemple	Résultat
<code>or</code>	OU logique	<code>(5 < 3) or (4 < 6)</code>	<code>True</code>
<code>not</code>	NON logique	<code>not (5 > 3)</code>	<code>False</code>

✦ Opérations sur les chaînes de caractères

Opération	Description	Exemple	Résultat
<code>+</code>	Concaténation	<code>"Hello " + "World"</code>	<code>"Hello World"</code>
<code>*</code>	Répétition	<code>"Python" * 3</code>	<code>"PythonPythonPython"</code>

✦ Opérations d'affectation combinées

Opérateur	Équivalent à	Exemple	Résultat
<code>+=</code>	<code>x = x + 3</code>	<code>x += 3</code>	<code>x</code> augmente de 3
<code>-=</code>	<code>x = x - 2</code>	<code>x -= 2</code>	<code>x</code> diminue de 2
<code>*=</code>	<code>x = x * 4</code>	<code>x *= 4</code>	<code>x</code> est multiplié par 4
<code>/=</code>	<code>x = x / 2</code>	<code>x /= 2</code>	<code>x</code> est divisé par 2
<code>//=</code>	<code>x = x // 2</code>	<code>x //= 2</code>	<code>x</code> est divisé entier par 2
<code>%=</code>	<code>x = x % 2</code>	<code>x %= 2</code>	<code>x</code> devient le reste de la division par 2
<code>**=</code>	<code>x = x ** 2</code>	<code>x **= 2</code>	<code>x</code> est élevé au carré

✦ Opérations de concaténation

`"a"+"b" --> "ab"`

✦ Structures de contrôle (conditions if/else, boucles for et while)

✅ **Conditions (if/else):** Les conditions permettent d'exécuter un bloc de code seulement si une condition est vraie.

```
x = 10
if x > 5:
    print("x est supérieur à 5")
elif x == 5:
    print("x est égal à 5")
else:
    print("x est inférieur à 5")
```

🔄 **Boucle for:** La boucle for est utilisée pour parcourir une séquence (liste, chaîne, range...)

```
for i in range(5): # De 0 à 4
    print(i)
```

```
fruits = ["pomme", "banane", "cerise"]
```

```
for fruit in fruits:
    print(fruit)
```

ou

```
[fruit for fruit in fruits]
```

 **Boucle while:** La boucle while répète un bloc tant qu'une condition est vraie

```
x = 0
while x < 5:
    print(x)
    x += 1
```

Fonctions et modules

Python permet d'organiser et de réutiliser du code grâce aux **fonctions** et aux **modules**.

Les Fonctions en Python

Une **fonction** est un bloc de code réutilisable qui effectue une tâche spécifique. Elle peut prendre des **paramètres** et renvoyer une **valeur**. Sans retour, on parle des fois de **procédure**

Déclaration et appel d'une fonction

$$f(x) = x + 2$$

```
In [20]: def f(x):
         return x+2

         f(2)
```

Out[20]: 4

```
In [21]: g = lambda x: x+2
         g(2)
```

Out[21]: 4

```
In [22]: def saluer(nom):
         """Cette fonction affiche un message de salutation."""
         print(f"Bonjour, {nom} !")

         # Appel de la fonction
         saluer("Alice")
```

Bonjour, Alice !

Fonction avec retour de valeur

```
In [ ]: def carre(x):
         """Renvoie le carré du nombre x."""
         return x ** 2

         resultat = carre(4)
         print(resultat) # Affiche 16
```

16

Paramètres avec valeurs par défaut

```
In [25]: def saluer(nom="Inconnu"):
          print(f"Bonjour, {nom} !")

          saluer() # Affiche "Bonjour, Inconnu !"
          d = saluer("Bob") # Affiche "Bonjour, Bob !"
```

Bonjour, Inconnu !

Bonjour, Bob !

Les Modules en Python

Un module est un fichier contenant du code Python (fonctions, classes, variables) que l'on peut importer et utiliser dans un autre programme.

Importation d'un module standard

```
In [27]: import math
          print(math.sqrt(16)) # Affiche 4.0
```

4.0

Importation partielle

```
In [4]: from math import sqrt
          print(sqrt(25)) # Affiche 5.0
```

5.0

Création et importation d'un module personnalisé

```
In [28]: import os
          os.getcwd()
```

Out[28]: 'd:\\Projet\\formation\\src'

```
In [6]: os.chdir("/home/kamoussou/Kokou/Projet/formation/src")
```

```
In [ ]: from modules.utils import dire_bonjour
          print(dire_bonjour("Alice"))
```

Bonjour, Alice !

```
In [1]: from modules.utils import carre
          print(carre(4))
```

16

```
In [8]: import modules.utils as u
          print(u.dire_bonjour("Alice"))
```

Bonjour, Alice !

Notions de POO (Introduction aux classes et objets)

Un objet : une voiture, une personne, un compte bancaire, un animal, etc

La **Programmation Orientée Objet (POO)** est un paradigme qui organise le code en objets. Un objet est une instance d'une **classe**, qui regroupe des **attributs** (données) et des **méthodes** (fonctions associées).

✓ Définition d'une classe et création d'un objet

```
In [ ]: class Personne:
    """Classe représentant une personne."""

    def __init__(self, nom, age):
        """Constructeur qui initialise les attributs nom et âge."""
        self.nom = nom
        self.age = age

    def sePresenter(self):
        """Méthode pour afficher une présentation."""
        return f"Bonjour, je m'appelle {self.nom} et j'ai {self.age} ans."

# Création d'un objet (instance de la classe)
personne1 = Personne("Alice", 25)

# Appel d'une méthode
print(personne1.sePresenter())
```

Bonjour, je m'appelle Alice et j'ai 25 ans.

🔒 Encapsulation (Attributs privés)

L'**encapsulation** protège les attributs d'une classe en les rendant privés (**préfixe `__` ou `_`**).

```
In [2]: class CompteBancaire:
    def __init__(self, titulaire, solde):
        self.titulaire = titulaire
        self.__solde = solde # Attribut privé

    def deposer(self, montant):
        """Ajoute un montant au solde."""
        self.__solde += montant

    def retirer(self, montant):
        self.__solde -= montant

    def afficher_solde(self):
        """Affiche le solde du compte."""
        return f"Solde de {self.titulaire} : {self.__solde} €"

# Création d'un compte
compte = CompteBancaire("Bob", 1000)
compte.deposer(500)
print(compte.afficher_solde())
print(compte.__solde)
```

Solde de Bob : 1500 €

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[2], line 21
    19 compte.deposer(500)
    20 print(compte.afficher_solde())
--> 21 print(compte.__solde)

AttributeError: 'CompteBancaire' object has no attribute '__solde'
```

Héritage (Créer une classe enfant)

L'**héritage** permet de créer une nouvelle classe basée sur une autre.

```
In [ ]: class Animal:
    def __init__(self, nom):
        self.nom = nom

    def parler(self):
        """Méthode générique à surcharger."""
        return "Je fais un bruit."

# Classe Chien qui hérite de Animal
class Chien(Animal):
    def parler(self):
        return "Ouaf !"

# Classe Chat qui hérite de Animal
class Chat(Animal):
    def parler(self):
        return "Miaou !"

# Création des objets
chien = Chien("Rex")
chat = Chat("Mimi")

print(chien.nom, "dit", chien.parler())
print(chat.nom, "dit", chat.parler())
```

```
Rex dit Ouaf !
Mimi dit Miaou !
```


Polymorphisme

Le **polymorphisme** permet d'utiliser une même méthode avec des comportements différents selon la classe.

```
In [14]: animaux = [Chien("Buddy"), Chat("Felix")]

for animal in animaux:
    print(animal.nom, "dit", animal.parler())
```

```
Buddy dit Ouaf !
Felix dit Miaou !
```

La **POO** permet d'organiser le code en objets réutilisables, facilitant la maintenance et l'évolutivité. 

⇒ [Revenir au déroulé](#)