


PYTHON






1. PRESENTATION

1.1 Histoire rapide

 **Créateur** : *Guido van Rossum*, Pays-Bas NL




 **Date de création** : 1989 (officiellement publié en 1991)

 **Objectif initial** : Proposer un langage simple et lisible pour automatiser des tâches système

 Anecdote : Nommé "Python" en hommage aux *Monty Python*, pas au serpent
 !

1.2 Philosophie du langage : « batteries included »

🧠 C'est une expression qui résume la philosophie de Python : "tout ce qu'il faut est déjà inclus".

-  Livré avec une **bibliothèque standard très complète**
-  Permet de manipuler fichiers, réseau, données, UI, maths... sans installer de packages tiers
-  Approche orientée vers la **productivité rapide**









🔧 Ex :

Module	Utilité
math	Fonctions mathématiques (sin, cos, sqrt...)
random	Générer du hasard (utile pour nos générateurs 3D 🎲)
datetime	Gérer les dates et les heures
os	Interagir avec le système (lister des fichiers, lire des dossiers)
json	Lire/écrire des fichiers JSON
re	Rechercher avec des expressions régulières

1.3 Le Zen de Python



19 principes qui guident la conception du langage :

 Principe	 Explication
 "Beautiful is better than ugly."	Le code doit être élégant et clair , pas bricolé ou confus.
 "Simple is better than complex."	Vise la simplicité même si une solution plus complexe est possible.
 "Readability counts."	La lisibilité du code est primordiale . On écrit pour être lu par d'autres humains, pas juste pour la machine.
 "There should be one-- and preferably only one --obvious way to do it."	Il vaut mieux avoir une façon claire et évidente de faire les choses, plutôt que plusieurs ambiguïtés.
 "Now is better than never."	Agir (même imparfaitement) vaut mieux que procrastiner.
 "Errors should never pass silently."	Les erreurs doivent être visibles et compréhensibles , pas ignorées.



En résumé:

Le Zen de Python promeut un **code lisible, simple et explicite**.



Priorité à la **clarté**



Rejet des **abstractions inutiles**



Favorise la **maintenance** du code



Utile pour des scripts **collaboratifs** ou réutilisables (ex: outils Blender)

1.4 Pourquoi Python est populaire ?

✓ Avantage	🔍 Détail
📄 Syntaxe claire	Moins de code pour faire plus
🔗 Écosystème riche	Des milliers de bibliothèques utiles
💻 Multiplateforme	Fonctionne sous Windows, macOS, Linux
👥 Communauté active	Beaucoup de tutoriels, forums et support
🎨 Utilisé dans la 3D	Compatible avec Blender, Maya, Houdini...

🔍 En résumé:

Python = Simple, puissant et extensible.

- ✓ Parfait pour les débutants ET pour les professionnels
- ✓ Très utilisé en **3D, animation, VFX, IA, web, data science...**

2. PYTHON ET LA 3D

🔧 Pourquoi apprendre Python dans la modélisation 3D ?

2.1 Scripting = Automatiser les tâches répétitives

🚫 Fin des clics manuels à répétition

🧠 Un script peut :

- Ajouter automatiquement des matériaux
- Générer des objets paramétriques (ex: escaliers, pièces modulaires)
- Exporter/importer des fichiers de façon personnalisée

🕒 **Gain de temps énorme** pour la production et la prévisualisation


2.2 Python dans les pipelines 3D = le liant

 Python est utilisé pour **relier** plusieurs logiciels dans un pipeline de production :


➤ Blender → Substance → Maya → Unreal Engine

✨ Exemple :

Un script peut exporter une scène Blender + générer un fichier JSON + l'envoyer automatiquement dans un moteur de jeu

 Permet une **intégration fluide**, sans passer par des clics manuels ni conversions manuelles







2.3 Créer ses propres outils (tooling)

 Avec Python, tu peux développer :

- Des **UI personnalisées** dans Blender
- Des **générateurs procéduraux**
- Des **validateurs de scène** (ex : vérifier qu'il n'y a pas de mesh sans nom)

 Tu deviens **acteur technique du studio**, pas juste utilisateur

2.4 Logiciels 3D qui supportent Python

Logiciel	Intégration Python
 Blender	API Python native bpy – très complète
 Maya	Scriptable en Python (remplace MEL)
 Houdini	Python dans les nodes et pour automatiser
 3ds Max	Supporte Python via MaxScript
 Cinema 4D	API Python moderne (R23+)
 Unity & Unreal	Utilisable dans les outils de production, pipeline, etc.



En résumé:

Python = **clé de voûte technique** dans les studios d'animation, VFX, jeu vidéo, architecture...

- ✓ Crée tes outils
- ✓ Automatise ta production
- ✓ Intègre dans un pipeline pro

3.2 Créer un environnement virtuel

Un environnement virtuel permet d'isoler les bibliothèques d'un projet.

🔒 Idéal pour **éviter les conflits entre projets**.

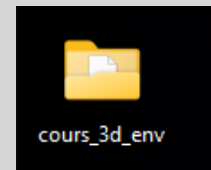
➤ Toujours dans un terminal on choisit un dossier de destination (pour l'exemple sur le bureau)

```
cd Desktop
```

➤ Ensuite on crée un environnement virtuel

```
python -m venv cours_3d_env
```

📁 Cela crée un dossier « cours_3d_env » contenant une copie locale de Python.



On doit maintenant activer l'environnement virtuel

```
cours_3d_env\Scripts\activate
```

N.B: il se peut que windows nous bloque,

il faut donc passer la commande

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

**Pour désactiver l'environnement
Dans le terminal:
deactivate**

```
PS C:\Users\jul89\Desktop> cours_3d_env\Scripts\activate  
(cours_3d_env) PS C:\Users\jul89\Desktop> |
```

4. CONFIGURATION DE VS CODE

4.1 Pourquoi VS Code?

- 🆓 Gratuit, open-source et léger
- ⚙️ Compatible avec tous les OS (Windows, macOS, Linux)
- 🧠 Excellente intégration avec Python (débugueur, linting, IntelliSense)

4.2 Extension à installer

🧩 Extension	💬 Description
🐍 Python (Microsoft)	Interpréteur, IntelliSense, debug, exécution de scripts
📄 Python Docstring Generator	Génère des commentaires de documentation automatiques (""" """)
🔍 Pylint	Vérifie la qualité du code (style, erreurs, bonnes pratiques PEP8)

✅ Ces outils vous aideront à **coder proprement et efficacement**.

4.3 Configuration de l'environnement Python dans VS Code

- 1: Ouvrir son projet avec VS Code
- 2: Ouvrir la **palette de commande** (Ctrl + Shift + P)
- 3: Taper: Python: Select Interpreter
- 4: Choisir votre environnement virtuel : cours_3d_env\Scripts\python.exe

⚠ Cette étape est **essentielle** pour exécuter vos scripts dans le bon contexte !





4.4 Installer les packages de base pour vos futurs projets 3D

Dans le terminal intégré de VS Code (Ctrl + ou menu *Terminal* → *Nouveau terminal*) :

On va exécuter 2 commandes:

```
pip install --upgrade pip
```


```
pip install numpy matplotlib
```

 Package	 Utilité
 matplotlib	Pour faire des visualisations simples (graphiques, courbes)
 numpy	Pour manipuler facilement des tableaux, utile en 3D (vecteurs, matrices)

4.5 Astuce pour les librairies supplémentaires

 Astuce : Crée un fichier : requirements.txt

À l'intérieur on répertorie les librairies que l'on souhaite installer






```
requirements.txt
1  numpy
2  matplotlib
3
```

on peut donc réinstaller toutes les librairies en une commande

`pip install -r requirements.txt`

 Pratique pour **partager ton projet avec d'autres** ou pour ton portfolio.

 **Bonus : autres extensions utiles (facultatif)**

-  **autoDocstring** : autre générateur de docstring, très complet
-  **Code Runner** : permet d'exécuter un script sans config
-  **Material Icon Theme** : améliore la lisibilité de l'arborescence

5. BASE DE PROGRAMMATION

5.1 Les variables en Python

« Des boîtes pour stocker des valeurs »




Qu'est-ce qu'une variable ?

Étiquette qui pointe vers une **valeur** en mémoire.

Créée au moment de l'affectation :

```
x = 42  
nom = "Alice"
```

	Détail
Typage dynamique	Le type dépend de la valeur ; il peut changer.
Pas de déclaration	Pas de <code>int x</code> ; - la création est implicite.
Sensibles à la casse	Age \neq age
Référence, pas conteneur	Deux variables peuvent pointer vers le même objet.

Règles de nommage (PEP 8)


Lettres, chiffres, `_`, **jamais** d'espace ni de caractère spécial.


Ne pas commencer par un chiffre.

Style conseillé : **snake_case**

```
user_score = 100
```

Exemple	Type	Code
42	entier (int)	age = 42
3.14	flottant (float)	pi = 3.14
"Hello"	chaîne (str)	msg = "Hello"
True	booléen (bool)	is_ready = True
[1, 2]	liste (list)	coords = [1, 2]

 Pour connaître le type d'une variable : `type(variable)` => `type(user_score)`

 Changer de valeur, changer de type

Python l'autorise, mais  restez cohérent pour éviter les bugs !






```
n = 10      # int
n = "dix"   # devient str
```

5.2 Les tableaux numériques

Définition

Un **tableau** (en Python, on parle de **liste**) est une collection **ordonnée et modifiable** d'éléments.

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

Action	Code Python	Résultat
 Accéder à un élément	fruits[0]	"pomme"
 Longueur	len(fruits)	3
 Ajouter un élément	fruits.append("kiwi")	["pomme", ..., "kiwi"]
 Supprimer un élément	fruits.remove("banane")	["pomme", "cerise"]
 Parcourir	for fruit in fruits:	affiche chaque fruit

À retenir

- ✓ Indice commence à 0
- ✓ Une liste peut contenir **différents types** (int, str, float, etc.)
- ✓ Les listes sont **mutables** (modifiables)

```
mix = [1, "texte", 3.14]
```

5.3 Les tableaux associatifs

Définition

Un **tableau associatif** (ou **dictionnaire**) permet de stocker des **paires clé → valeur**.

```
personne = {  
    "nom": "Alice",  
    "âge": 30,  
    "ville": "Paris"  
}
```

À retenir

- ✓ Les clés sont uniques (pas de doublons)
- ✓ Les valeurs peuvent être de n'importe quel type
- ✓ On utilise les {} pour définir un dictionnaire

Action	Code Python	Résultat
 Accéder à une valeur	personne["nom"]	"Alice"
 Nombre d'éléments	len(personne)	3
 Ajouter / modifier	personne["email"] = "alice@mail.com"	Ajoute une nouvelle entrée
 Supprimer une clé	del personne["ville"]	Supprime "ville"
 Parcourir (clé + valeur)	for k, v in personne.items():	Affiche chaque clé/valeur

5.4 Les opérateurs

« Manipuler et comparer des valeurs »



1. Opérateurs arithmétiques

Symbole	Signification	Exemple Python	Résultat
+	Addition	<code>3 + 2</code>	5
-	Soustraction	<code>7 - 4</code>	3
*	Multiplication	<code>5 * 6</code>	30
/	Division	<code>10 / 2</code>	5.0
//	Division entière	<code>10 // 3</code>	3
%	Modulo (reste)	<code>10 % 3</code>	1
**	Puissance	<code>2 ** 3</code>	8



Utilisés pour faire des calculs simples ou complexes.

5.4 Les opérateurs



2. Opérateurs de comparaison

Symbole	Signification	Exemple	Résultat
==	Égal à	5 == 5	True
!=	Différent de	3 != 4	True
>	Supérieur à	7 > 2	True
<	Inférieur à	1 < 9	True
>=	Supérieur ou égal	4 >= 4	True
<=	Inférieur ou égal	5 <= 2	False



Renvoient un booléen (True ou False) – utilisés dans les conditions

5.4 Les opérateurs

3. Opérateurs logiques

Symbole	Signification	Exemple	Résultat
and	Et logique	True and False	False
or	Ou logique	True or False	True
not	Négation logique	not True	False

 Permettent de combiner ou d'inverser des conditions.

4. Opérateurs d'affectation

Symbole	Signification	Exemple	Équivaut à
=	Affectation	x = 5	—
+=	Ajoute puis assigne	x += 2	x = x + 2
-=	Soustrait puis assigne	x -= 1	x = x - 1
*=	Multiplie puis assigne	x *= 3	x = x * 3
/=	Divise puis assigne	x /= 2	x = x / 2

 Renvoient un booléen (True ou False) – utilisés dans les conditions

5.4 Les opérateurs

5. Autres opérateurs utiles

Type	Exemple	Résultat
Appartenance	'a' in 'chat'	True
Identité	a is b	True si a et b pointent le même objet
Concaténation	"Hello" + " world"	"Hello world"
Répétition	"ha" * 3	"hahaha"

À retenir :

Les opérateurs permettent de **manipuler** et **tester** des données.

Ils sont souvent utilisés dans des **expressions conditionnelles** (if, while, etc.)

Bien comprendre leur **type de retour** (ex: True, False, float, int, etc.)

5.5 Les conditions

«Permettre au programme de **prendre des décisions**.»

1. Condition if / else

 Exemple concret : Si l'utilisateur a plus de 18 ans, il peut accéder au contenu.

```
âge = 20

if âge >= 18:
    print("Accès autorisé")
else:
    print("Accès refusé")
```



Rappel important :

Indentation obligatoire (4 espaces)

Les blocs **if / else** ne **prennent pas de parenthèses**

5.5 Les conditions

«Permettre au programme de **prendre des décisions**.»

2. Condition multiple avec elif

```
note = 14

if note >= 16:
    print("Très bien")
elif note >= 12:
    print("Assez bien")
else:
    print("À revoir")
```



elif = else if

Permet de tester plusieurs cas

- ✓ Utiliser des noms de variables clairs
- ✓ Bien indenter le code
- ✓ Ne pas empiler trop de conditions dans un seul if
- ✓ Préférer elif à if + else imbriqués

On peut aussi utiliser une condition raccourcie:

```
âge = 16
autorisation = "Oui" if âge >= 18 else "Non"
print(authorisation) # Affiche "Non"
```

5.5 Les conditions

«Permettre au programme de **prendre des décisions**.»

2. Depuis Python 3.10+ : Structure match / case (nouveau)

```
choix = "a"

match choix:
    case "a":
        print("Option A")
    case "b":
        print("Option B")
    case _:
        print("Option inconnue")
```

- ✓ Disponible à **partir de Python 3.10**
- ✓ Très lisible, proche du switch des autres langages
- ✓ Permet de tester beaucoup de cas

5.6 Les boucles

« Répéter des actions automatiquement »

1. Boucle for

Parcourt une séquence (liste, chaîne, range, etc.)

```
for i in range(3):  
    print("Tour", i)
```

Exemple :

Parcourir une liste d'éléments

```
fruits = ["pomme", "banane", "cerise"]  
for fruit in fruits:  
    print(fruit)
```


5.6 Les boucles



2. Boucle while

Répète tant qu'une condition est vraie

```
compteur = 0
while compteur < 3:
    print("Compteur =", compteur)
    compteur += 1
```



À noter pour la boucle while:

On déclare une variable (compteur = 0)

Dans le while on donne une condition (compteur < 3)

On n'oublie pas d'incrémenter la variable à chaque passage de boucle (compteur += 1)

5.6 Les boucles

3. Contrôle de boucle

Mot-clé	Description	Exemple
break	Sortir immédiatement de la boucle	if i == 2: break
continue	Passer à l'itération suivante sans exécuter le reste	if i == 2: continue

Mini-exercice rapide

```
for i in range(5):  
    if i == 3:  
        break  
    print(i)
```

Affiche 0, 1, 2 puis stoppe la boucle.

Points clés

- ✓ **for** : idéal pour parcourir une collection ou un intervalle.
- ✓ **while** : utile quand la condition de fin dépend d'un calcul.
- ✓ **break & continue** permettent de mieux contrôler la boucle.
- ✓ Indentation obligatoire pour définir le bloc à répéter.

5.7 Les fonctions



Introduction aux Fonctions



Qu'est-ce qu'une fonction ?

- Bloc de code **réutilisable**
- Permet de **structurer** et **factoriser** un programme
- Peut recevoir des **paramètres** et retourner un **résultat**



Avantages :

- Moins de duplication de code
- Plus facile à lire, maintenir et tester
- Permet de diviser un problème complexe en sous-problèmes

5.7 Les fonctions



Déclaration d'une Fonction



Syntaxe de base:

```
def nom_de_la_fonction(param1, param2):  
    # Instructions  
    return resultat
```



Exemple :

```
def saluer(nom):  
    print(f"Bonjour {nom} !")  
  
saluer("Alice") # Affiche : Bonjour Alice !
```



Une fonction peut être appelée **autant de fois que nécessaire**

5.7 Les fonctions



Retour de Valeur & Paramètres



Fonction avec retour

```
def addition(a, b):  
    return a + b  
  
resultat = addition(2, 3)  
print(resultat) # Affiche : 5
```



Paramètres par défaut

```
def bonjour(nom="visiteur"):  
    print(f"Bonjour {nom} !")  
  
bonjour() # Bonjour visiteur !  
bonjour("Lucie") # Bonjour Lucie !
```



Lorsqu'on donne une valeur par défaut, on n'est pas obligé d'en mettre un lors de l'appel

5.7 Les fonctions

Fonctions intégrées & Docstring

🌟 Fonctions Python les plus courantes :

Fonction	Rôle
print()	Affiche un message
input()	Lit une saisie clavier
len()	Renvoie la taille
type()	Renvoie le type
int(), str()	Conversion de types



Comment documenter une fonction ?

```
def multiplier(x, y):  
    """Multiplie deux nombres et renvoie le résultat"""  
    return x * y
```

5.7 Les fonctions

Bonnes Pratiques et Concepts Avancés

Bonnes pratiques :


- Donner un nom clair à vos fonctions
- Une fonction = une responsabilité
- Toujours commenter avec des docstrings

Concepts avancés (à découvrir) :

- Fonctions anonymes : lambda
- Portée des variables (global, local)
- Fonctions imbriquées
- Fonctions comme arguments

5.7 Les fonctions

Bonus - Fonctions Lambda & Portée des Variables

-  Fonctions anonymes (lambda)
- Fonction courte, sans nom
- Utilisée pour des opérations simples, en une ligne

```
# Fonction classique
def carre(x):
    return x * x

# Lambda équivalente
carre_lambda = lambda x: x * x
print(carre_lambda(4)) # Résultat : 16
```

✓ Souvent utilisée avec map(), filter(), ou sorted() :

```
nombres = [5, 2, 9]
triés = sorted(nombres, key=lambda x: -x)
print(triés) # [9, 5, 2]
```


5.7 Les fonctions



Portée des variables (scope)



Définit où une variable peut être utilisée. Fonction courte, sans nom

Type	Description
local	Déclarée dans une fonction
global	Déclarée en dehors de toute fonction
nonlocal	Utilisée dans une fonction imbriquée

Exemples:

```
x = 10 # Variable globale

def fonction():
    x = 5 # Variable locale
    print(x)

fonction()      # Affiche 5
print(x)        # Affiche 10 (global)
```



Utiliser global avec précaution :

```
x = 1

def increment():
    global x
    x += 1





increment()
print(x) # Résultat : 2
```

6. CONCLUSION

Ce que nous avons appris :

- ✓ Python : langage clair, puissant et accessible
- ✓ Variables, types, opérateurs
- ✓ Structures de contrôle : conditions, boucles
- ✓ Fonctions : définition, paramètres, retour
- ✓ Structures de données : listes, dictionnaires
- ✓ Cas pratiques adaptés à des besoins réels (ex. médias, 3D...)

Et maintenant ?

-  Approfondir les modules et bibliothèques standards
-  Créer des scripts utiles pour automatiser vos tâches
-  Explorer l'intégration Python dans vos outils métiers (ex. Blender, traitement de fichiers, etc.)
-  Structurer vos projets : fichiers, imports, fonctions réutilisables