Scripting Python

Ahmed MAME

Mis à jour : 30/06/2025

Scripting

- Le mot **« scripting »** est employé surtout dans le monde de l'informatique.
- Décrit le fait d'écrire des « scripts », des petits programmes (souvent interprétés plutôt que compilés).
- Destiné à automatiser des tâches, coller des briques logicielles entre elles ou piloter un logiciel plus vaste.
- **Automatisation** : renommer en masse des fichiers, extraire des données d'un tableur, faire clignoter une LED sur un microcontrôleur, etc.
- **Langages interprétés** : langages dits « de **scripting** » : Python, JavaScript, Bash, PowerShell, Ruby…

Scripting vs Programmation

- Tout scripting est de la programmation.
- Toute programmation n’est pas du scripting.
- Les langages compilés (C, C++, Rust, Go…) sont souvent choisis pour des logiciels « lourds » ou des systèmes embarqués.
- Programmation : La vitesse d’exécution et la gestion fine de la mémoire.
- Le scripting : La rapidité de développement et la flexibilité.
- Echanger un peu de performance contre du gain de temps et de confort.

Utilisation du scripting

Domaine	Exemple de scripts courants
Administration système	Automatisation des sauvegardes, déploiements, surveillance.
Développement web	Build des projets, dynamiser une page de navigateur web.
Analyse de données	Notebooks Python (pandas, NumPy) pou chargement, nettoyage et visualisation de données.
Jeux vidéo	Lua ou Python intégrés dans un moteur pour définir le comportement d'un PNJ ou un niveau.
Logiciels métiers	Macros VBA dans Excel ou scripts Python qui pilotent Blender, Photoshop, AutoCAD, etc.

Scripting en Python

- Un simple fichier texte dont l'extension est .py.
- Contient du code exécuté ligne après ligne par l'interpréteur python (ou python3).
- Souvent autonome : on le lance, il fait son travail puis se termine.
- Par opposition à un programme plus gros structuré en paquets, modules, tests, etc.
- Scripts réutilisables si bien structurés.
- Manipulation des fichiers et gestion des arguments/options.

Bonnes pratiques en Scripting python

Objectif	Bon réflexe	Pourquoi ?
Options	argparse, click ou typer	Gestion d'aide (-h) et validation d'entrée
Logs	logging au lieu de print partout	Choix du niveau (info, warn, error) et redirection dans un fichier
Fichiers	pathlib.Path plutôt que os.path	API orientée objet, cross-platform
Erreurs	try/except ciblés, messages clairs	Un script qui plante sans explication est frustrant
Réutilisabilité	Mettre la « logique » dans des fonctions	Import; Tests unitaires

Python – Langage de programmation

- Créé en 1990, Open source (PSF License)
- Interprété, haut-niveau, multiparadigme
- Lisibilité du code et simplicité
- Scripts système, Data science, IA/ML, Web (Django/Flask/FastAPI), Automatisation, IoT, Jeux, etc.
- Syntaxe claire : indentation obligatoire
- Écosystème gigantesque : plus de 400 000 paquets sur PyPI
- Exemples: TensorFlow ou PyTorch (IA); Django ou Flask (Web); …

Python – Installation

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.12 python3.12-venv python3.13-dev
```

Python – Environnements virtuels

- Un environnement virtuel est un dossier isolé contenant :
  - Sa propre copie (ou lien symbolique) de l'exécutable Python.
  - Un répertoire site-packages/ où seront installées uniquement les bibliothèques nécessaires à ce projet.
- Chaque projet peut avoir ses dépendances et versions spécifiques.
- Pas de risque de conflit avec d'autres projets ou avec les paquets installés globalement sur la machine.
- venv est le module intégré à Python (depuis la version 3.3) qui permet de créer des environnements virtuels.

Pourquoi utiliser venv ?

Besoin	Comment venv aide ?
Éviter les conflits de versions	Deux projets peuvent utiliser des versions différentes d'une même bibliothèque.
Reproductibilité	Le fichier requirements.txt + venv assurent qu'un collègue ou un serveur CI installe exactement les mêmes versions.
Simplicité de nettoyage	Supprimez simplement le dossier du venv pour tout désinstaller.
Droits limités	Pas besoin d'être root/administrateur : toutes les installations se font dans le dossier du projet.

Comment s’en servir ?

- Créer le venv (Linux/macOS) :

```
python3 -m venv .venv
```

- Activer l’environnement :

```
source .venv/bin/activate
```

- Installer les dépendances :

```
pip install flask requests
```

- Geler les versions :

```
pip freeze > requirements.txt
```

- Désactiver l’environnement :

```
deactivate
```

Un premier exemple

```
# -------- hello.py --------
# Ce script affiche simplement un message dans la console.

print("Salut, Python !")          # ← instruction exécutée

# Fin du script !
```

Les variables

- **Déclaration implicite** : on assigne, Python crée la variable et infère le type.
- **Typage dynamique** : une même variable peut changer de type (à éviter pour la lisibilité).
- **Mutabilité** : listes & dictionnaires sont modifiés « en place », pas les entiers ou chaînes.
- **Bon réflexe** : nommer tes variables en **snake_case** et garder un seul type par variable.

Les variables

```python
prenom    = "Alice"       # str
annee     = 2025          # int
taux_tva = 0.20           # float
tags      = ["python", "débutant"]  # list (mutable)

print(f"{prenom} - type : {type(prenom).__name__}")
```

Les Fonctions

- **Définition** :

```
def nom(param1, param2=valeur_defaut):
```

- **Portée (scope)** : tout ce qui est créé à l'intérieur est local sauf si déclaré global.
- **Docstring** : triple guillemets juste après def → utile pour l'aide en ligne (help()).
- **Lambda** ? Expressions anonymes d'une ligne :

```
carre = lambda x: x**2
```

Les Fonctions

```python
def aire_cercle(rayon, pi=3.14159):
    """Calcule l'aire d'un cercle (πr²)."""
    return pi * rayon ** 2          # **  = puissance

print(aire_cercle(3))               # 28.27...
```

Conditions

- Structure : if … elif … else. L'indentation fait partie de la syntaxe.
- Truthiness : 0, "", [], None sont évalués comme False.
- Opérateurs : and, or, not, comparaison chaînée (0 < x < 10).
- Astuce : préférer == pour l'égalité, is (réservé à l'identité d'objet).

```
age = 17
if age >= 18:
    statut = "majeur"
elif age >= 13:
    statut = "ado"
else:
    statut = "enfant"
print(statut)
```

Boucles

Boucle	Pour quoi faire ?	Particularités
for	Parcourir *séquence* ou *itérable*	range, enumerate, boucle else
while	Répéter jusqu'à condition fausse	Attention aux boucles infinies

- break sort immédiatement de la boucle, continue saute à l'itération suivante.

Boucles

```python
# for + range
for i in range(1, 4):           # 1, 2, 3
    print(i, aire_cercle(i))

# while
compteur = 0
while compteur < 3:
    print("Tour", compteur)
    compteur += 1

# Parcourir une liste avec index et valeur
courses = ["lait", "œufs", "pain"]
for idx, article in enumerate(courses, start=1):
    print(f"{idx}. {article}")
```

Listes

- **Création** : crochets [], mélange de types autorisé.
- **Indexation** : commence à 0 ; indices négatifs reprennent depuis la fin.
- **Mutabilité** : on peut modifier, ajouter ou retirer des éléments sans recréer la liste.
- **Méthodes clés** : append, extend, insert, pop, sort, reverse.
- **Astuce** : pour copier une liste (éviter la référence partagée), utiliser ma_liste.copy() ou le slicing ma_liste[:].

Listes

```
fruits = ["pomme", "banane", "cerise"]
print(fruits[0])                    # "pomme"
fruits.append("datte")              # ajoute en fin
fruits[1] = "myrtille"              # remplace "banane"
dernier = fruits.pop()              # retire et renvoie "datte"
print(fruits)                       # ['pomme', 'myrtille', 'cerise']
```

Listes

- **Création** : crochets [], mélange de types autorisé.
- **Indexation** : commence à 0 ; indices négatifs reprennent depuis la fin.
- **Mutabilité** : on peut modifier, ajouter ou retirer des éléments sans recréer la liste.
- **Méthodes clés** : append, extend, insert, pop, sort, reverse.
- **Syntaxe condensée** :

```
carrés = [x**2 for x in range(1, 6)]   # [1, 4, 9, 16, 25]
pairs  = [x for x in carrés if x % 2 == 0]
```

Dictionnaires

- **Utilité** : pour l'association rapide clé-valeur (≈ « table de hachage »).
- **Création** : accolades {cle: valeur}.
- **Clés** uniques et **hachables** (str, int, tuple immuable…).
- Accès/ajout/remplacement via dict[cle].
- **Parcours** par défaut sur les clés ; méthodes .items(), .values().
- **Méthodes utiles** : get (évite KeyError), pop, update, setdefault.

Dictionnaires

```python
etudiant = {"nom": "Alice", "age": 20, "notes": [15, 18, 14]}
print(etudiant["nom"])           # Alice
etudiant["age"] += 1             # anniversaire
etudiant["ville"] = "Paris"      # ajout d'une nouvelle clé

# Parcourir proprement
for cle, valeur in etudiant.items():
    print(f"{cle}: {valeur}")
```

Fichiers

- Lire un fichier texte ligne par ligne (encodage UTF-8 conseillé)

```
from pathlib import Path

chemin = Path("poeme.txt")      # équivalent à open('poeme.txt')
with chemin.open(encoding="utf-8") as f:
    for ligne in f:             # itération paresseuse → faible mémoire
        print(ligne.rstrip())   # rstrip supprime le \n final
```

- Lire tout le contenu d'un coup

```
contenu = chemin.read_text(encoding="utf-8")
print(contenu[:100], "…")       # affiche les 100 premiers caractères
```

Fichiers

- Écrire ou ajouter

```
# Écriture (écrase le fichier si déjà présent)
with open("log.txt", "w", encoding="utf-8") as f:
    f.write("Première ligne\n")

# Ajout en fin de fichier
with open("log.txt", "a", encoding="utf-8") as f:
    f.write("Ligne suivante\n")
```

- Lire un CSV rapidement (module standard)

```
with open("ventes.csv", newline="", encoding="utf-8") as f:
    lecteur = csv.DictReader(f)        # chaque ligne → dict
    total = 0
    for ligne in lecteur:
        total += float(ligne["montant"])
print("Total des ventes :", total)
```

Python et DevOps

Atout	Pour un DevOps ?
Syntaxe lisible + faible « boilerplate »	On écrit un POC ou un script d'automatisation en quelques minutes.
Écosystème pléthorique	Bibliothèques pour (presque) chaque cloud, chaque API, chaque outil d'infra.
Communauté & docs	Des exemples et des best-practices partout.
Domaines	IaC, CI/CD, API, Docker & K8S, Monitoring, Tests, Sécurité, Serverless, etc.

YAML : Le bloc-note du DevOps

- **IaC** : Toutes les ressources Kubernetes se décrivent en YAML.
- **Ansible** : Inventaires statiques ou dynamiques + Playbooks.
- **GitHub Actions** ou **GitLab CI** : le pipeline est déclaré dans un seul fichier YAML.
- **Templates & rendu** : Helm, Kustomize, Jinja2, Kapitan, etc. prennent du YAML modèle + des valeurs et génèrent du YAML final, facilitant la gestion de plusieurs environnements (dev/integration/prod).
- **Limites** : Indentation fragile, pièges sur les types ("true" ≠ true), pas de schéma imposé (d'où l'utilité des linters : kubeval, cfn-lint, yamllint).

JSON : la monnaie d’échange des outils

- **API First** : Tout ce qui passe sur HTTP échange généralement du JSON ; un script Python peut donc consommer directement les réponses (json.loads()).
- **Logs & observabilité** : Output de *Docker*, Fluent Bit/Logstash, OpenTelemetry exportent du JSON ; la stack Elastic le re-parse facilement.
- **États & rapports** : Fichiers terraform.tfstate ou SBOM CycloneDX sont en JSON pour être versionnés puis digérés par d’autres outils.
- **Sécurité** : Les scanners de sécurité (Trivy, Checkov) publient des résultats JSON, qu’un step CI convertit ensuite en HTML ou en badge.
- **Limites** : pas de commentaires, objets verbeux ; peu pratique à éditer manuellement pour de longs documents.

Exercices

Avant Pratique

- **PyYAML** : lecture/écriture rapide (mais écrase les commentaires).
- **ruamel.yaml** : conserve l'ordre, les commentaires et le style – pratique pour les manifests K8S.
- **jsonschema** : validation de données JSON.
- **jinja2** : templating avancé.

```
pip install pyyaml ruamel.yaml jsonschema jinja2
```

Lire & écrire un YAML simple

- Charger le fichier YAML deployment.yaml
- Incrémenter spec.replicas
- Sauvegarder le fichier sous le nom deployment_scaled.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo
```

Conversion YAML ↔ JSON

- Écrire une fonction yaml_to_json(path_yaml, path_json)
- Écrire une fonction json_to_yaml(path_json, path_yaml)

- Tests :

```
yaml_to_json("deployment.yaml", "deployment.json")

json_to_yaml("deployment.json", "deployment_roundtrip.yaml")
```

Fusion de manifests (overlay)

- On dispose des fichiers deployment.yaml base et d'un overlay.yaml contenant uniquement les champs à surcharger.
- Écrire un script qui produit un YAML fusionné.
- Il s'agit d'une « **merge-key update** » récursive, équivalente à un kubectl kustomize simplifié.

```yaml
spec:
  replicas: 5
  template:
    spec:
      containers:
      - name: demo
        image: demo:2.0
```

Validation JSON Schema

- Valider le contenu d'un fichier config.json contre le schéma schema.json.
- Le schéma :
  - https://json-schema.org/draft/2020-12/schema
- Utiliser jsonschema.
- Afficher la liste des erreurs avec le chemin exact de chaque champ non conforme.

```
{
  "serviceName": "demo",
  "replicas": 2,
  "image": "demo:1.0",
  "ports": [
    80,
    443
  ]
}
```