Programmation en Python - Introduction

Hervé TALE KALACHI

ENSPY – Université de Yaoundé 1

Public visé

- Étudiants MSP 2 (Licence 2)
- Avoir des bases algorithmiques
- Être motivé

Contenu du cours

- Mode d'utilisation de python
- Structures de données
- Structures de contrôle
- Création de fonctions
- Gestion des fichiers
- Erreurs et exceptions

Supports

- Cours de Python par Patrick Fuchs et Pierre Poulain
- Slides/notes de cours
- Fiches de TD/TP

Le langage Python (haut niveau $\leftrightarrow$ bas niveau)

Haut niveau = langage avec *fortes abstractions* proches du problème :

- Structures de données riches (listes, dictionnaires, ensembles), itérateurs.
- Gestion mémoire automatique.
- Typage dynamique et syntaxe concise $\Rightarrow$ prototypage rapide.
- Exécution via interpréteur/bytecode $\Rightarrow$ tests immédiats (REPL).

Bas niveau (ex. C) = langage plus proche de la machine :

- Contrôle fin de la mémoire (pointeurs, `malloc/free`).
- Typage statique explicite, compilation/édition de liens.
- Performances d'exécution souvent supérieures $\Rightarrow$ coût de dev. plus élevé.

À retenir : Python = *productivité, lisibilité, portabilité*. C (et similaires) = *contrôle, performance, proximité matériel*.

Haut niveau vs bas niveau : repères pratiques

Haut niveau (Python)

- Abstractions riches, syntaxe concise
- Gestion mémoire automatique
- Typage dynamique (prototypage rapide)
- Portabilité multi-OS (scripts souvent inchangés)
- Cycle rapide (REPL, scripts)

Bas niveau (C, etc.)

- Proche du matériel, contrôle fin
- Mémoire manuelle (pointeurs)
- Typage statique (contrats à la compilation)
- Recompilation selon la cible/OS
- Performances optimisées

Implication pour ce cours : priorité à la clarté, la compréhension des concepts et l'expérimentation rapide.

Qu'est-ce que le REPL ?

REPL = Read–Eval–Print Loop (*Lire–Évaluer–Afficher*).

- Interface interactive de Python (prompt >>>).
- Permet de tester des idées et obtenir un retour immédiat.
- Idéal pour explorer les fonctions, types et bibliothèques.
- Nous l'utiliserons plus tard pour nos premières manipulations.

Compilation vs Interprétation — définitions

Compilation

- Traduction *avant* exécution : **code source** $\rightarrow$ **binaire natif**.
- Étapes : analyse, optimisation, génération d'exécutable.
- Cible spécifique (OS/architecture).
- L'exécutable tourne *sans* le compilateur.

Interprétation

- Traduction *à la volée* : l'interpréteur lit et exécute directement.
- Feedback immédiat (REPL).
- Forte portabilité (même source, interpréteur adapté).
- Nécessite l'interpréteur pour s'exécuter.

Idée clé : compilation = *prépayer* le coût de traduction ; interprétation = *payer au fil de l'eau*.

Chaîne d'exécution : où passe le code ?

Modèle compilé (ex. C/C++)
Source → Compilateur → Binaire/Libs → OS/CPU

Modèle interprété (général)
Source → Interpréteur → Exécution directe

Cas Python (courant)

- *Source* `.py` → **bytecode** `.pyc` (cache `__pycache__`) → **machine virtuelle Python** (interpréteur) → exécution.
- Le bytecode est *portable* entre OS (même version d'interpréteur).

NB : Python n'est donc pas « binaire natif » par défaut ; il s'appuie sur son interpréteur pour exécuter le bytecode.

Avantages / Inconvénients et choix pragmatique

Compilation

- + Performances d'exécution élevées.
- + Déploiement sans interpréteur.
- − Cycle build plus lent (compile → lier → exécuter).
- − Portage = recompiler selon la cible.

Interprétation

- + Itération rapide, REPL, prototypage.
- + Portabilité et simplicité de distribution du `.py`.
- − Overhead à l'exécution (généralement plus lent).
- − Dépendance à l'interpréteur installé.

Règle pratique :

→ *Cours, prototypage, data, automatisation* : interprété (Python).

→ *Temps réel, embarqué, calcul intensif critique* : compilé (ou hybride).

Cas hybrides et écosystème Python

- **Hybride compilation+interprétation** : certains langages compilent vers un *bytecode* exécuté par une VM (ex. Java). Python suit un schéma proche (bytecode $\rightarrow$ VM Python).
- **Accélération ciblée** en Python (aperçu) :
  - Implémentations alternatives avec JIT (ex. VM avec compilation à la volée).
  - Extensions natives (C/C++), bibliothèques optimisées (*sous le capot*, code compilé).
  - Outils de « gel »/packaging (création d'exécutables pour distribution).
- **Pour ce cours** : on reste côté *interprétation* + *bytecode*, focalisé sur la *clarté* et la *pratique*.

Installation de Python

- **Télécharger** depuis `https://www.python.org` (version 3.x récente).
- **Windows** : installer Python 3, cocher *Add Python to PATH*.
- **macOS** : utiliser l'installateur officiel (ou `brew install python`).
- **Linux** : Python 3 déjà présent dans la plupart des distributions ; sinon via le gestionnaire de paquets.
- **Outils utiles** : `pip` (gestion de paquets), un éditeur/IDE (VS Code, PyCharm, Thonny).