Session 0 : Setup de la Stack de Développement

Introduction

1 Objectifs de cette session

1. Installer et configurer Visual Studio Code comme IDE principal
2. Configurer un environnement Python adapté à son OS
3. Installer et comprendre Git pour le contrôle de version
4. Installer uv comme gestionnaire de packages
5. Installer Gemini CLI

CLI, shell et bash (définitions)

Un shell est un programme qui agit comme un intermédiaire entre l'utilisateur et l'OS. C'est le shell qui nous permet de taper des commandes pour qu'elles soient process et que l'output nous soit renvoyé et affiché dans le CLI. Bash est le plus connu des shells pour systèmes Linux et souvent le par défaut. Le terme CLI (command line interface) fait référence à l'interface textuelle où l'utilisateur intéragit en tapant les commandes, contrairement au shell qui est le programme d'interprétation des commandes en soit.

2 Partie 1 : Installation de Visual Studio Code

2.1 Pourquoi VS Code ?

Définition : IDE (Integrated Development Environment)

Un IDE est un environnement de développement intégré qui combine plusieurs outils essentiels au développement logiciel dans une seule application : éditeur de code avec coloration syntaxique, débogueur, terminal intégré, et extensions pour personnaliser l'expérience. Contrairement à un simple éditeur de texte, un IDE comprend le contexte de votre code et peut vous aider à naviguer, refactorer et déboguer efficacement.

L'avantage de VScode est d'être gratuit, open source et avec beaucoup d'extensions intéressantes mais rien ne vous empêche d'en utiliser un autre type Pycharm.

2.2 Installation

Suivre les instructions : `https://code.visualstudio.com/download`

2.3 Extensions intéressantes à installer

Ouvrez VS Code, puis accédez aux extensions (`Ctrl+Shift+X` ou `Cmd+Shift+X`) et installez :
Pour Windows uniquement :

Extension	ID	Utilité
Python	`ms-python.python`	Support Python complet
Pylance	`ms-python.vscode-pylance`	Analyse statique et IntelliSense
Python Debugger	`ms-python.debugpy`	Débogage Python
GitLens	`eamodio.gitlens`	Visualisation Git avancée
Ruff	`charliermarsh.ruff`	Linter et formatter ultra-rapide

Extension	ID	Utilité
WSL	`ms-vscode-remote.remote-wsl`	Développement dans WSL
Dev Containers	`ms-vscode-remote.remote-containers`	Conteneurs de développement

3 Partie 2 : Configuration de l'Environnement Python

Pour ceux sous Windows, je vous conseille de suivre la démarche pour installer (ou activer) WSL2 et utiliser docker, c'est un peu overkill mais en même vraiment bien et cela vous pousse à vous habituer à un environnement "pro". Pour Mac, je ne connais pas spécialement donc vous pouvez checker des tutos ou simplement setup VScode et les extensions.

3.1 Configuration Windows avec WSL

Définition : WSL (Windows Subsystem for Linux)

WSL est une couche de compatibilité développée par Microsoft permettant d'exécuter un environnement Linux directement sur Windows, sans machine virtuelle traditionnelle. WSL2 utilise un véritable noyau Linux et offre des performances quasi-natives.

Définition : Dev Container

Un Dev Container (conteneur de développement) est un environnement de développement complet encapsulé dans un conteneur Docker. Il définit toutes les dépendances, outils et configurations nécessaires au projet. L'avantage majeur est la reproductibilité : chaque développeur travaille dans un environnement identique, éliminant les problèmes du type "ça marche sur ma machine".

Pourquoi WSL + Dev Containers pour Windows ?

- **Compatibilité** : La plupart des outils de data sont développés pour Unix/Linux
- **Isolation** : Chaque projet peut avoir son propre environnement sans conflits
- **Reproductibilité** : Votre environnement est défini dans du code, partageable avec l'équipe
- **Professionnalisme** : C'est une approche utilisée dans le monde pro (d'après ma courte expérience)

3.1.1 Étape 2.A.1 : Installation de WSL2

Suivez les instructions décrites ici : https://learn.microsoft.com/fr-fr/windows/wsl/install

Ouvrez PowerShell **en tant qu'administrateur** et exécutez :

```
# Installation de WSL avec Ubuntu (distribution par defaut)
wsl --install

# Redemarrez votre ordinateur apres cette commande
```

Après le redémarrage, Ubuntu se lancera automatiquement pour finaliser l'installation. Créez un nom d'utilisateur et un mot de passe Unix.

Vérification :

```
wsl --version
```

3.1.2 Étape 2.A.2 : Configuration initiale de Ubuntu

Ouvrez Ubuntu depuis le menu Démarrer et exécutez :

```
# Mise a jour du systeme
sudo apt update && sudo apt upgrade -y

# Installation des dependances essentielles
sudo apt install -y build-essential curl wget git

# Verification de Python (pre-installe sur Ubuntu)
python3 --version
```

3.1.3 Étape 2.A.3 : Installation de Docker Desktop

Définition : Docker

Docker est une plateforme de conteneurisation qui permet d'empaqueter une application avec toutes ses dépendances dans une unité standardisée appelée conteneur. Contrairement aux machines virtuelles, les conteneurs partagent le noyau de l'OS, ce qui les rend légers et rapides à démarrer.

Téléchargement : https://www.docker.com/products/docker-desktop/

1. Téléchargez et installez Docker Desktop pour Windows
2. Pendant l'installation, assurez-vous que l'option "Use WSL 2 instead of Hyper-V" est cochée
3. Après l'installation, ouvrez Docker Desktop
4. Allez dans Settings → Resources → WSL Integration
5. Activez l'intégration pour votre distribution Ubuntu

Vérification dans WSL :

```
docker --version
docker run hello-world
```

3.1.4 Étape 2.A.4 : Développer dans WSL

Suivez les instructions décrites ici : https://code.visualstudio.com/docs/remote/wsl

Créer votre premier projet avec Dev Container :

Suivez les instructions décrites ici : https://code.visualstudio.com/docs/remote/wsl#_advanced-opening-a-wsl-2-folder-in-a-container

4 Partie 3 : Installation et Introduction à Git

Définition : Git

Git est un système de contrôle de version. Il permet de suivre l'historique des modifications de votre code, de collaborer avec d'autres développeurs, et de revenir à des versions antérieures si nécessaire.

Définition : Repository (Dépôt)

Un repository (ou "repo") est un espace de stockage pour votre projet qui contient tous les fichiers du projet ainsi que l'historique complet de leurs modifications. Un repo peut être local (sur votre machine) ou distant (sur GitHub, GitLab, etc.). Le repo distant sert de sauvegarde et de point de synchronisation pour la collaboration.

Définition : Commit

Un commit est un instantané de votre projet à un moment donné. Chaque commit a un identifiant unique (hash SHA-1), un message descriptif, et référence le commit parent. Les commits forment une chaîne qui constitue l'historique de votre projet. Un bon commit est atomique (une seule modification logique) et bien documenté.

Pourquoi Git est essentiel ?

- **Historique complet** : Chaque modification est tracée avec auteur, date et raison
- **Branches** : Développez des fonctionnalités en parallèle sans risque
- **Collaboration** : Travaillez en équipe sur le même code
- **Sauvegarde** : Votre code est sécurisé sur des serveurs distants
- **Reproductibilité** : Reconstruisez l'état exact de votre code à n'importe quel moment
- **Standard industriel** : Utilisé par toutes les entreprises qui développent un minimum

4.1 Installation de Git

Suivez les instructions décrites ici : https://git-scm.com/install/
Git sur WSL : https://learn.microsoft.com/en-us/windows/wsl/tutorials/wsl-git (pas besoin d'aller jusqu'a setup le credentials manager)

4.2 Configuration initiale de Git

```
# Identite
git config --global user.name "Votre Nom"
git config --global user.email "votre.email@example.com"

# Branche par defaut
git config --global init.defaultBranch main

# Verification de la configuration
git config --list
```

4.3 Commandes Git essentielles

```
# Initialiser un nouveau repository
git init

# Cloner un repository existant
git clone <url>

# Voir l'etat des fichiers
git status

# Ajouter des fichiers au staging
git add <fichier>
git add .  # Tous les fichiers
```

```
# Creer un commit
git commit -m "Description de la modification"

# Voir l'historique
git log --oneline

# Creer une branche
git branch <nom-branche>
git checkout -b <nom-branche>  # Creer et basculer

# Basculer entre branches
git checkout <nom-branche>
git switch <nom-branche>  # Syntaxe moderne

# Fusionner une branche
git merge <nom-branche>

# Synchroniser avec le distant
git push origin <branche>
git pull origin <branche>
```

4.4 Création d'un compte GitHub

URL : https://github.com/join

4.5 Configuration de l'authentification SSH (Pas obligatoire)

```
# Generer une cle SSH avec algo rsa
ssh-keygen -t rsa -C "votre.email@example.com"

# Demarrer l'agent SSH
eval "$(ssh-agent -s)"

# Ajouter la cle a l'agent
ssh-add ~/.ssh/id_ed25519

# Copier la cle publique
cat ~/.ssh/id_ed25519.pub
```

Allez sur GitHub → Settings → SSH and GPG keys → New SSH key, et collez votre clé.

Secure Shell (SSH) est un protocole utilisé pour accéder de manière sécurisée à des systèmes à distance. Dans le contexte data science, c'est un concept que l'on retrouve souvent pour pouvoir exécuter des programmes sur des serveurs à distance, accéder à du stockage cloud ou gérer des machines virtuelles. Ce protocole permet donc le transfert sécurisé des données souvent sensibles avec lesquelles on peut travailler. C'est un outil qui peut être intéressant à comprendre/creuser en fonction de votre proximité au dev/déploiement dans votre futur métier.

Test de la connexion :

```
ssh -T git@github.com
```

5 Partie 4 : Installation de uv (Gestionnaire de Packages)

Définition : Gestionnaire de Packages

Un gestionnaire de packages est un outil qui automatise l'installation, la mise à jour, la configuration et la suppression de bibliothèques logicielles. En Python, les gestionnaires de packages résolvent les dépendances (si le package A nécessite le package B version 2.0+, le gestionnaire s'en occupe), gèrent les conflits de versions, et isolent les environnements pour éviter les incompatibilités entre projets.

Définition : Environnement Virtuel

Un environnement virtuel Python est un répertoire isolé contenant une installation Python spécifique et ses packages. Chaque projet peut avoir son propre environnement avec ses propres versions de packages, évitant les conflits. Par exemple, un projet de backtesting peut utiliser pandas 1.5 tandis qu'un autre projet utilise pandas 2.0, sans interférence.

Pourquoi uv plutôt que pip ou conda ?

- **Vitesse exceptionnelle** : 10-100x plus rapide que pip grâce à une implémentation en Rust
- **Résolution de dépendances moderne** : Algorithme de résolution plus intelligent
- **Gestion des environnements intégrée** : Pas besoin d'outils séparés
- **Compatibilité pip** : Utilise les mêmes packages que pip (PyPI)
- **Lockfile** : Garantit la reproductibilité exacte des installations
- **Moderne** : Développé par Astral (créateurs de Ruff), adopté rapidement par la communauté

Documentation officielle : https://docs.astral.sh/uv/

5.1 Installation de uv

Windows (dans WSL) :

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Ajouter au PATH (si nécessaire) :

```
# Pour bash
echo 'export PATH="$HOME/.cargo/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

# Pour zsh (macOS par defaut)
echo 'export PATH="$HOME/.cargo/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

Vérification :

```
uv --version
```

5.2 Utilisation de uv

Cheat sheet cool pour UV : https://mathspp.com/blog/uv-cheatsheet

5.2.1 Initialiser un nouveau projet

```
# Creer un nouveau projet Python
uv init mon-projet
cd mon-projet

# Structure creee :
# mon-projet/
#           .python-version
#           README.md
#           hello.py
#           pyproject.toml
```

5.2.2 Créer un projet de type librairie

```
uv init pyvest --lib --package
cd pyvest

# Structure creee :
# pyvest/
#           .python-version
#           README.md
#           pyproject.toml
#           src/
#               pyvest/
#                   __init__.py
```

5.2.3 Gérer l'environnement virtuel

```
# Creer un environnement virtuel
uv venv -p 3.12 .venv

# Activer l'environnement (Linux/macOS)
source .venv/bin/activate

# Activer l'environnement (Windows PowerShell dans WSL)
# Note: Dans WSL, utilisez la commande Linux ci-dessus

# Desactiver l'environnement
deactivate
```

5.2.4 Installer des packages

```
# Installer un package
uv add numpy

# Installer plusieurs packages
uv add pandas matplotlib scipy

# Installer avec version specifique
uv add "numpy>=1.24,<2.0"

# Installer des dependances de developpement
uv add --dev pytest ruff mypy

# Synchroniser depuis pyproject.toml
uv sync
```

5.2.5 Fichier pyproject.toml typique

```toml
[project]
name = "pyvest"
version = "0.1.0"
description = "Librairie Python pour la gestion de portefeuille"
readme = "README.md"
requires-python = ">=3.12"
dependencies = [
    "numpy>=1.24",
    "pandas>=2.0",
    "scipy>=1.11",
    "matplotlib>=3.7",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.0",
    "ruff>=0.1",
    "mypy>=1.0",
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[tool.ruff]
line-length = 88
target-version = "py312"

[tool.ruff.lint]
select = ["E", "F", "I", "N", "W"]
```

6 Partie 5 : Installation de Gemini CLI

Ce tuto fonctionne pas mal pour WSL : https://www.linkedin.com/pulse/from-zero-hero-your-ultimate-guide-installing-gemini-cli-slobodian-4jbof/

Documentation officielle : https://github.com/google-gemini/gemini-cli

6.1 Prérequis : Installation de Node.js

Gemini CLI nécessite Node.js (runtime JavaScript).

6.2 Configuration de Gemini CLI

1. **Obtenir une clé API Gemini :**
   - Allez sur https://aistudio.google.com/apikey
   - Connectez-vous avec votre compte Google
   - Créez une nouvelle clé API
2. **Configurer la clé API :**

```bash
# Ajouter la cle a votre environnement
# Pour bash (WSL)
echo 'export GEMINI_API_KEY="votre-cle-api"' >> ~/.bashrc
source ~/.bashrc

# Pour zsh (macOS)
echo 'export GEMINI_API_KEY="votre-cle-api"' >> ~/.zshrc
source ~/.zshrc
```

3. **Vérifier l'installation :**

```
gemini --version
gemini "Explique-moi ce qu'est un DataFrame pandas en une phrase"
```

6.3 Alternative : Utilisation sans WSL (Windows natif)

Si vous choisissez de ne pas utiliser WSL sur Windows :

1. **Installer Node.js pour Windows :**
   - Téléchargez depuis https://nodejs.org/
   - Choisissez la version LTS
   - Exécutez l'installateur
2. **Ouvrir PowerShell et installer Gemini CLI :**

```
npm install -g @google/gemini-cli
```

3. **Configurer la variable d'environnement :**

```
# Temporaire (session courante)
$env:GEMINI_API_KEY = "votre-cle-api"

# Permanent (via System Properties)
# Panneau de configuration -> Systeme -> Parametres systeme avances
# -> Variables d'environnement -> Nouvelle variable utilisateur
# Nom: GEMINI_API_KEY
# Valeur: votre-cle-api
```