

Scripts d'exploitation

Tout ce dont vous avez besoin pour ce module, dans un seul document. Cochez au fur et à mesure.

U6-SP2-M4

module

Élève

document

TechnoVert

scénario 2

B · 3 h

format et durée

Ce qu'il faut savoir avant de commencer	4
Comment lancer un script Python dans PowerShell ?	4
Comment rattraper une erreur avec try et except ?	4
C'est quoi une base SQLite ?	4
Comment écrire un fichier CSV en Python ?	5
Vérifiez-vous	5
Écrire un script exploitable	6
Capsule 1 sur 4 Un script qu'on règle sans toucher au code	6
Capsule 2 sur 4 Dire si ça a marché : code de retour et journal	8
Capsule 3 sur 4 Lancer le script à heure fixe	10
Capsule 4 sur 4 Sauvegarder, et prouver qu'on sait restaurer	12
Ce qu'on retient	15
Écrire un script exploitable	16
L'essentiel	16
Les mots à connaître	16
Les codes à connaître	17
La méthode : rendre un script prêt à tourner seul	17
Un exemple concret	17
Les pièges	18
Script d'inventaire, journal et restauration	19
PARTIE A — Un script réglé par ses arguments	20
PARTIE B — Un journal et des codes de retour	25
PARTIE C — Planifier le script	30
PARTIE D — Sauvegarder et restaurer la base	35
POUR ALLER PLUS LOIN	39
CE QU'IL FAUT RETENIR	40
Tous les documents du module	41
A1 — Le dossier de travail du labo et son mode d'emploi	41
A2 — Extrait de la documentation de Python	45
A3 — Les règles d'exploitation des scripts de MaisonDesJeux	48

Ce qu'il faut savoir avant de commencer

Ce qu'il faut savoir avant de commencer. Lisez-le si vous avez un doute. Environ 10 minutes.

Comment lancer un script Python dans PowerShell ?

Un fichier Python se lance avec la commande `python`, suivie du nom du fichier.

```
python bonjour.py
```

PowerShell cherche le fichier dans le **dossier courant**. C'est le dossier où vous vous trouvez, écrit avant le `>` de l'invite.

La commande `ls` liste son contenu. La commande `cd` change de dossier.

Un chemin qui ne commence pas par une lettre de disque, comme `rapports\liste.csv`, part du dossier courant. On l'appelle un chemin relatif.

Comment rattraper une erreur avec try et except ?

Quand une instruction échoue, Python s'arrête et affiche une trace. Avec `try` et `except`, on prévoit ce qu'il faut faire à la place.

```
try:
    age = int("douze")
except ValueError:
    print("Ce n'est pas un nombre.")
```

Ici, `int("douze")` échoue. Python ne plante pas : il affiche le message.

C'est quoi une base SQLite ?

Une base SQLite tient dans un seul fichier, par exemple `club.db`. Elle contient des tables. Chaque table a des colonnes et des lignes.

En Python, le module `sqlite3` ouvre la base et envoie des requêtes SQL.

```
import sqlite3
con = sqlite3.connect("club.db")
nombre = con.execute("SELECT COUNT(*) FROM membre").fetchone()[0]
con.close()
```

`SELECT COUNT(*)` compte les lignes de la table `membre`.

Comment écrire un fichier CSV en Python ?

Un fichier CSV est un tableau écrit en texte : une ligne par ligne du tableau, les colonnes séparées par un point-virgule.

```
import csv
with open("liste.csv", "w", newline="", encoding="utf-8") as f:
    ecrivain = csv.writer(f, delimiter=";")
    ecrivain.writerow(["nom", "age"])
    ecrivain.writerow(["Durand", 16])
```

Le fichier obtenu contient deux lignes : `nom;age`, puis `Durand;16`.

Vérifiez-vous

-
- ☐ 1. PowerShell affiche `PS C:\Users\lea\Documents\club>`. Vous tapez `python liste.py`. Dans quel dossier doit se trouver `liste.py` ?
-
- ☐ 2. Que fait ce code : `try: x = 10 / 0` puis `except ZeroDivisionError: x = 0` ?
-
- ☐ 3. Quelle requête SQL compte les lignes de la table `jeu` ?
-

Réponses

1. Dans `C:\Users\lea\Documents\club`, le dossier courant.
2. La division par zéro échoue. Python ne plante pas : `x` prend la valeur 0.
3. `SELECT COUNT(*) FROM jeu;`

Écrire un script exploitable

COURS-U6-S1-07

Marc Oliveira est le trésorier de MaisonDesJeux, une association de jeux de société. Il tient le fichier des adhérents depuis six ans. Ce fichier a déjà été perdu deux fois.

Aujourd'hui, le fichier est une base de données sur le poste du bureau. Pour la protéger, Marc la copie sur une clé USB, quand il y pense.

Sa dernière copie date de quatre mois. Il ne l'a jamais ouverte.

Bruno, un bénévole, propose d'écrire « un petit programme qui fait ça tout seul ». Marc hésite : « Et s'il plante la nuit, qui me le dira ? »

À la fin de la séance, vous saurez écrire un script qui tourne seul et qui dit s'il a réussi. Et vous saurez prouver qu'une sauvegarde marche.

LE PLAN DE LA SÉANCE

- 1 Un script qu'on règle sans toucher au code.
- 2 Dire si ça a marché : code de retour et journal.
- 3 Lancer le script à heure fixe.
- 4 Sauvegarder, et prouver qu'on sait restaurer.

LA QUESTION DE DÉPART

Un programme tourne seul à 6 h du matin. Personne n'est devant l'écran. Il échoue.
Comment le savoir le lendemain ?

CAPSULE 1 SUR 4

Un script qu'on règle sans toucher au code

Bruno veut un programme qui fait une tâche à la place de Marc. Première question : en quoi est-ce différent des programmes que vous avez déjà écrits ?

On l'appelle un **SCRIPT**. C'est un petit programme qui fait une tâche d'exploitation à la place d'un humain : copier, compter, vérifier, exporter.

Il n'a pas de fenêtre. Il est souvent lancé par une machine, pas par une personne.

Personne n'est donc là pour répondre à un `input()`. Le script reçoit ses réglages au moment où on le lance. Ces réglages s'appellent des **ARGUMENTS** : des valeurs écrites sur la ligne de commande, après le nom du script.

Pensez à une machine à laver. Vous choisissez 30 ou 60 degrés à chaque lavage. Vous ne démontez pas la machine pour changer la température.

- 1 On écrit le **nom** de l'argument avec deux tirets : `--dossier`.
- 2 On écrit sa **valeur** juste après, séparée par une espace : `partage`.
- 3 Dans le script, le module **argparse** lit la ligne. Il range chaque valeur sous son nom : `args.dossier`.
- 4 Si un argument obligatoire **manque**, argparse affiche le mode d'emploi et arrête le script.

```
parser = argparse.ArgumentParser(description="Inventaire d'un dossier.")
parser.add_argument("--dossier", required=True, help="le dossier à inventorier")
parser.add_argument("--sortie", default="rapports", help="le dossier des rapports")
args = parser.parse_args()
```

`required=True` rend l'argument obligatoire. `default` donne la valeur utilisée quand l'argument est absent.

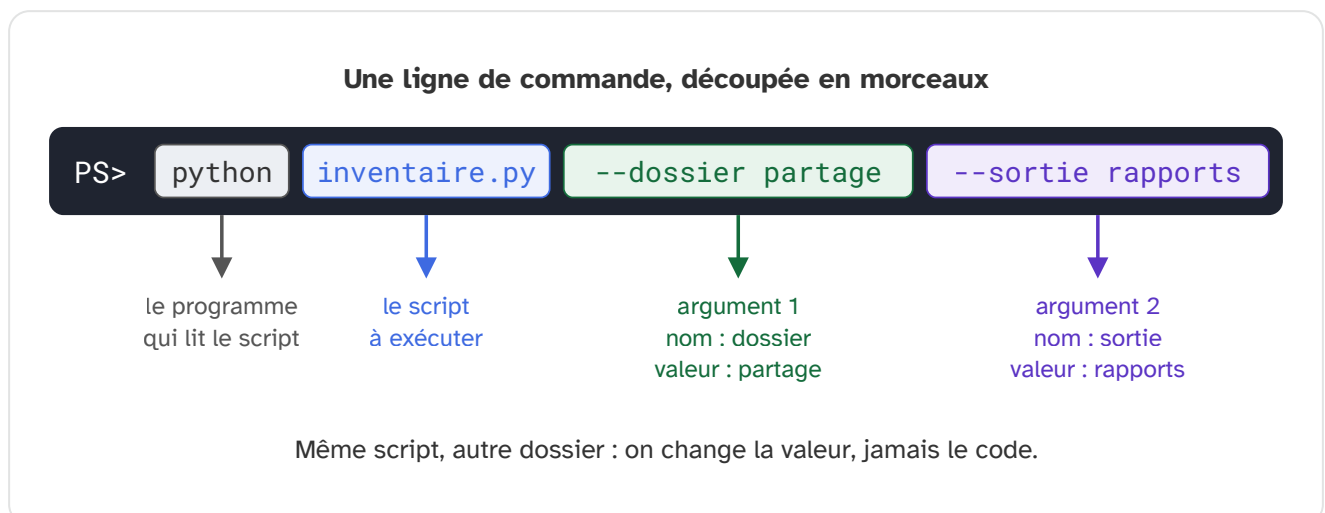


Schéma • Une ligne de commande découpée : le programme, le script, puis chaque argument avec son nom et sa valeur

ON LE FAIT ENSEMBLE — LA SAUVEGARDE DE MARC

1. Marc tape `python sauvegarde.py --base adherents.db --dest E:\copies`. Quel est le script ?
2. Combien d'arguments ?
3. Quelle est la valeur de `--dest` ?
4. Marc change de clé. Elle apparaît en `F:`. Que modifie-t-il ?

Exemple

Pourquoi

✓ <code>python inventaire.py --dossier partage\public</code>	Même script, autre dossier : seule la valeur change.
✗ Modifier <code>dossier = "partage"</code> dans le code avant chaque lancement	On risque de casser le script. Et une machine ne peut pas modifier le code à votre place.
✗ Demander le dossier avec <code>input()</code>	Le script qui tourne seul attend une réponse que personne ne tapera. Il reste bloqué.

CHECKPOINT

À l'oral, 3 minutes. Chacun note sa réponse sur sa feuille, puis on corrige.

1. On lance `python inventaire.py --dossier public --sortie bilan`. Quelle est la valeur de `args.sortie` ?
2. On lance `python inventaire.py --sortie bilan`. Que se passe-t-il, avec le code ci-dessus ?

CAPSULE 2 SUR 4

Dire si ça a marché : code de retour et journal

Le script se règle sans toucher au code. Reste la question de Marc : s'il échoue la nuit, comment le savoir ?

Le script le dit de deux façons.

D'abord, par son **CODE DE RETOUR**. C'est un nombre que le script rend au système quand il s'arrête : **0** si tout va bien, un autre nombre sinon. Le nombre dit quelle erreur a eu lieu.

Ensuite, par son **JOURNAL**. C'est un fichier texte où le script écrit ce qu'il fait, une ligne par événement, avec la date et l'heure.

Pensez au suivi d'un colis. L'appli affiche « Livré » ou « Échec de livraison » : c'est le code. Le détail dit « Absent, avis déposé à 14 h 32 » : c'est le journal.

- 1 Le script se termine par `sys.exit(0)` quand tout est fait.
- 2 Pour chaque échec prévu, il rend un **code choisi par l'équipe** : 3 pour « dossier introuvable », par exemple. Les codes sont écrits dans les règles de l'équipe.
- 3 Python rend **1** tout seul quand le script plante. argparse rend **2** quand les arguments sont faux.
- 4 Le code est lu par **une machine** : PowerShell le range dans `$LASTEXITCODE`, l'ordonnanceur le garde aussi.
- 5 Le journal est lu par **un humain**. Le module **logging** écrit chaque ligne avec son heure et son niveau : INFO pour une étape normale, ERROR pour un échec.

```
logging.info("Début de l'inventaire de %s", args.dossier)
if not dossier.is_dir():
    logging.error("Dossier introuvable : %s", dossier)
    sys.exit(3)
```

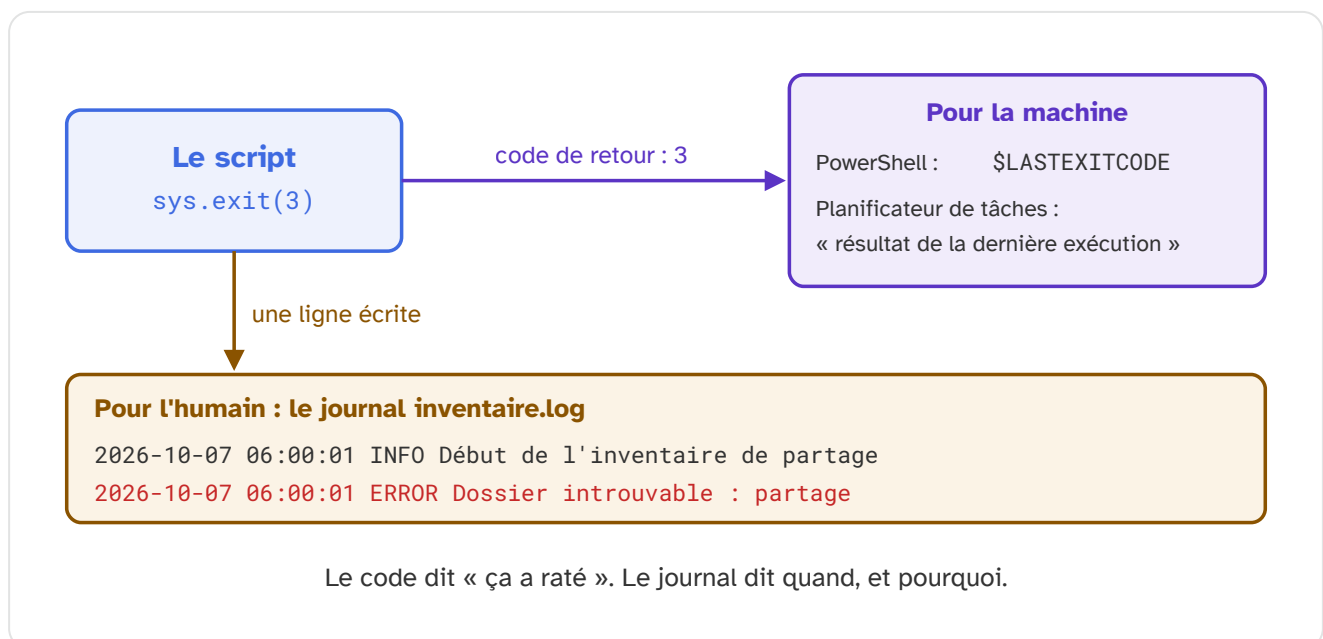


Schéma • Le code de retour part vers la machine, le journal garde la trace pour l'humain

ON LE FAIT ENSEMBLE — LA CLÉ USB OUBLIÉE

1. Marc lance la sauvegarde, mais la clé n'est pas branchée. Le script peut-il rendre 0 ?
2. Les règles de l'équipe disent 3 pour « dossier introuvable ». Quelle ligne termine le script ?
3. Que doit contenir le journal ?
4. Le script plante sur une erreur que personne n'a prévue. Quel code rend-il ?

Exemple

Pourquoi

✓	<code>sys.exit(0)</code> à la dernière ligne, quand tout est fait	Le code dit clairement : réussi.
✗	<code>print("Erreur !")</code> , puis le script se termine normalement	Le code vaut 0. La machine croit que tout va bien. Personne n'est prévenu.
✗	Une ligne de journal <code>Erreur</code> sans date ni cause	On ne sait ni quand ni pourquoi. Le journal ne sert à rien.

CHECKPOINT

À l'oral, 3 minutes. Chacun note sa réponse sur sa feuille, puis on corrige.

1. Après un lancement, `$LASTEXITCODE` vaut 2. Quelle est la cause la plus probable ?
2. Pourquoi écrire dans un fichier journal plutôt qu'à l'écran avec `print` ?

CAPSULE 3 SUR 4

Lancer le script à heure fixe

Le script dit maintenant s'il a réussi. Mais Marc doit encore penser à le lancer. Comment le faire partir tout seul ?

On confie ce travail à un **ORDONNANCEUR**. C'est un programme du système qui lance des tâches à l'heure prévue.

Sur Windows, c'est le Planificateur de tâches. Sur Linux, c'est cron.

C'est comme programmer le réveil de votre téléphone. Vous le réglez une fois. Il sonne chaque matin, même si vous dormez.

Une tâche planifiée a deux parties : le **déclencheur**, qui dit quand, et l'**action**, qui dit quoi lancer.

- 1 **Testez** le script à la main d'abord. Il doit marcher et rendre 0.
- 2 **Réglez le déclencheur** : une fois, chaque jour, chaque lundi, le 1er du mois.
- 3 **Réglez l'action** : le programme `python.exe`, les arguments, et le **dossier de départ**. Les chemins relatifs du script partent de ce dossier.
- 4 **Vérifiez une vraie exécution** : l'heure de la dernière exécution, son code de retour, et la ligne du journal à cette heure.

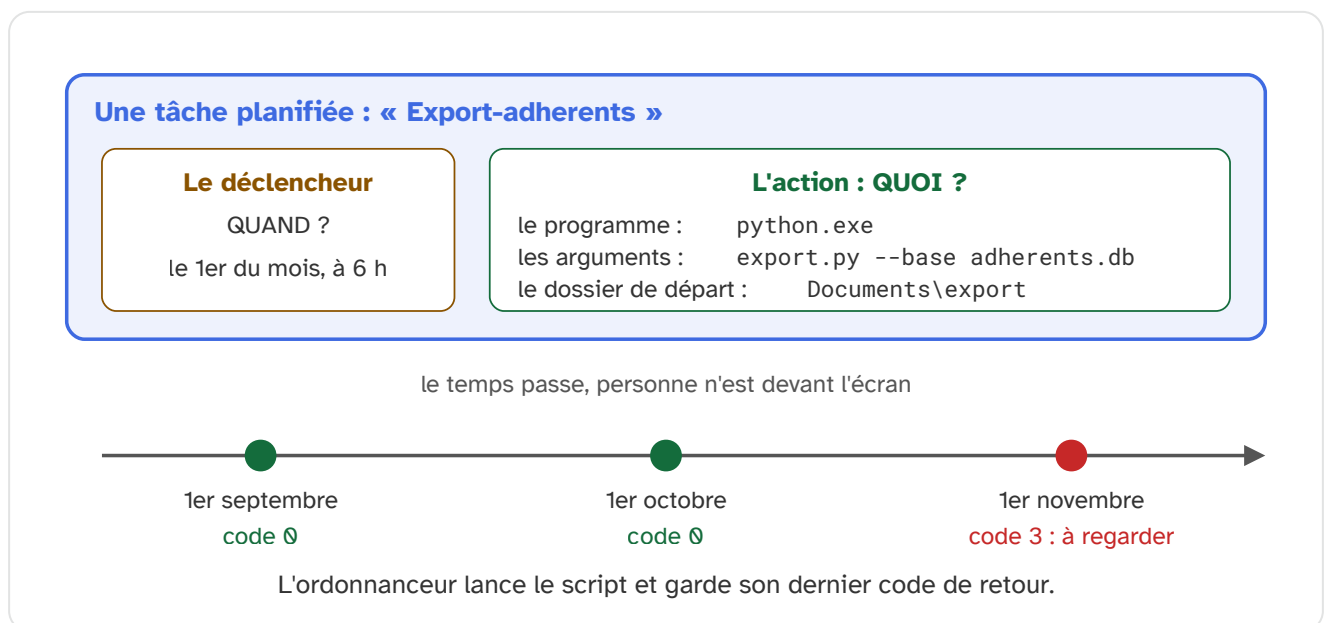


Schéma • Une tâche planifiée : un déclencheur, une action, et le dernier code de retour gardé à chaque exécution

ON LE FAIT ENSEMBLE — L'INVENTAIRE DU LUNDI

1. Marc veut l'inventaire du partage chaque lundi à 8 h. Quel est le déclencheur ?
2. Quel programme l'action lance-t-elle ?
3. Quel dossier de départ ?
4. Mardi, le Planificateur affiche 3 comme dernier résultat. Que faites-vous ?

Exemple	Pourquoi
✓ Lancer la tâche une fois à la main, puis lire son résultat et le journal	On voit une vraie exécution avant de partir.
✗ « La tâche est créée, donc ça marche. »	Créer n'est pas exécuter. Seule une exécution réelle, avec le code 0, le prouve.
✗ Planifier la tâche à 3 h sur un poste éteint la nuit	Un ordinateur éteint ne lance rien. Choisissez une heure où le poste est allumé.

CHECKPOINT

À l'oral, 3 minutes. Chacun note sa réponse sur sa feuille, puis on corrige.

1. Le Planificateur affiche 0 comme dernier résultat, à 6 h 00 ce matin. Qu'est-ce que ça prouve ?
2. Quelle erreur fait écrire le journal dans un autre dossier que prévu ?

CAPSULE 4 SUR 4

Sauvegarder, et prouver qu'on sait restaurer

On sait lancer un script seul, et savoir s'il a réussi. Revenons à la clé USB de Marc. Ses copies sont-elles vraiment utiles ?

Une **SAUVEGARDE** est une copie datée des données, rangée ailleurs, pour les retrouver après une perte. La **RESTAURATION**, c'est reconstruire les données à partir de cette copie.

Pour comparer deux bases, on calcule leur **EMPREINTE**. C'est un code court calculé à partir de tout le contenu. Si une seule lettre change, l'empreinte change.

Vos photos sont sauvegardées sur un cloud. Le jour où le téléphone tombe dans l'eau, vous les récupérez sur un nouveau téléphone. Mais vous ne savez si ça marche qu'en essayant.

INSTANT HISTOIRE — 1998 : TOY STORY 2 PRESQUE EFFACÉ

En 1998, chez Pixar, une commande d'effacement est lancée par erreur. Elle supprime la plupart des fichiers du film Toy Story 2.

Les sauvegardes ne marchaient plus depuis des semaines. Personne ne l'avait vu.

Le film est sauvé par une copie qu'une responsable gardait chez elle, sur son ordinateur.

Pixar avait des sauvegardes. Qu'est-ce qui a manqué ?

- 1 **Exportez** la base dans un fichier daté : `sauvegarde_2026-10-01.sql` , par exemple.
- 2 **Rangez** la copie ailleurs : un autre disque, un autre lieu. Jamais seulement à côté de la base.
- 3 **Restaurez** dans une **base neuve**, jamais sur la base en service.
- 4 **Comparez** avant et après : le nombre de lignes de chaque table, et l'empreinte. Tout doit être identique.

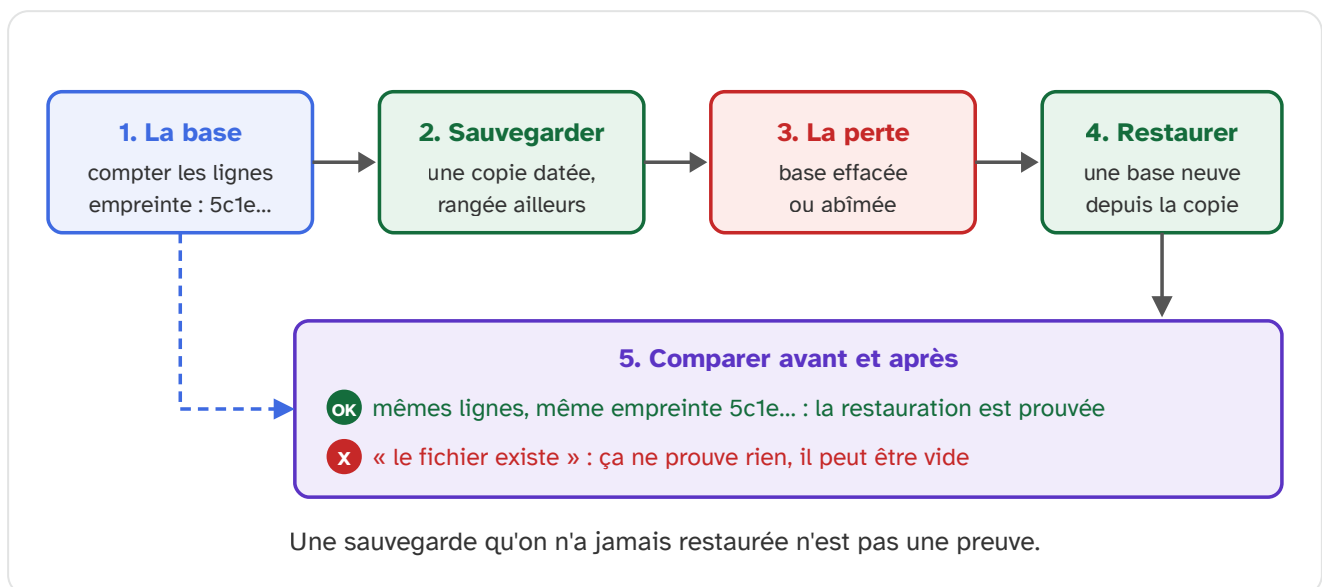


Schéma • Sauvegarder, perdre, restaurer, puis comparer : seule la comparaison prouve la restauration

ON LE FAIT ENSEMBLE — LA CLÉ USB DE MARC

1. La copie de Marc a quatre mois. Est-ce une sauvegarde ?
2. Où la restaurer pour la tester ?
3. Comment savoir si elle est complète ?
4. La copie a moins de lignes que la base actuelle. Pourquoi ?

Exemple

Pourquoi

✓	Chaque mois, restaurer la dernière copie dans un dossier de test et comparer	La sauvegarde est prouvée à chaque fois.
✗	Ranger la sauvegarde sur le même disque que la base	Si le disque lâche, la base et sa copie disparaissent ensemble.
✗	« Le fichier de sauvegarde pèse 180 Ko, donc il est bon. »	La taille ne dit rien du contenu. Seules la restauration et la comparaison le prouvent.

CHECKPOINT

À l'oral, 3 minutes. Chacun note sa réponse sur sa feuille, puis on corrige.

1. Pourquoi restaurer dans une base neuve, et pas sur la base en service ?
2. Avant et après la restauration, les tables ont le même nombre de lignes, mais l'empreinte d'une table est différente. La restauration est-elle prouvée ?

Ce qu'on retient

SI VOUS NE DEVEZ RETENIR QUE TROIS CHOSES

1. Un script se règle par ses arguments et rend un code de retour : 0 si tout va bien, un autre nombre sinon.
2. Un script qui tourne seul écrit un journal daté. On vérifie toujours une vraie exécution : son heure, son code, sa ligne de journal.
3. Une sauvegarde n'est prouvée que restaurée dans une base neuve, puis comparée à l'original.

Revenons à Marc. Un script exporte maintenant la base le 1er de chaque mois, à 6 h, lancé par le Planificateur. S'il échoue, il rend un code d'erreur et l'écrit dans son journal.

Et chaque mois, quelqu'un restaure la dernière copie dans un dossier de test. Marc ne se demande plus si ses copies marchent : il le voit.

Écrire un script exploitable

COURS-U6-S1-07

L'essentiel

- **Script** : un petit programme sans fenêtre, qui fait une tâche d'exploitation, souvent lancé par une machine.
- **Arguments** : les réglages donnés sur la ligne de commande. On change la valeur, jamais le code.
- **Code de retour** : 0 si tout va bien, un autre nombre sinon. Il est lu par une machine.
- **Journal** : une ligne datée par événement, lue par un humain.
- **Planification** : l'ordonnanceur lance le script à l'heure prévue et garde son dernier code de retour.
- **Sauvegarde** : prouvée seulement après une restauration dans une base neuve et une comparaison.

Les mots à connaître

Mot	En clair
Argument	Une valeur donnée au script sur la ligne de commande : <code>--dossier partage</code> .
argparse	Le module Python qui lit les arguments et affiche le mode d'emploi.
Code de retour	Le nombre rendu par le script quand il s'arrête. <code>\$LASTEXITCODE</code> dans PowerShell.
logging	Le module Python qui écrit le journal, avec l'heure et le niveau.
Ordonnanceur	Le programme qui lance les tâches à l'heure prévue : Planificateur de tâches, cron.
Dossier de départ	Le dossier d'où partent les chemins relatifs quand la tâche se lance.
Empreinte	Un code court calculé sur tout le contenu. Un seul changement, et elle change.

Les codes à connaître

Code	Qui le rend	Sens
0	le script, avec <code>sys.exit(0)</code>	tout va bien
1	Python, tout seul	le script a planté
2	argparse, tout seul	arguments faux ou manquants
3 et plus	le script, avec <code>sys.exit(...)</code>	un échec prévu, selon les règles de l'équipe

La méthode : rendre un script prêt à tourner seul

1. **Remplacez** les valeurs écrites dans le code par des arguments.
2. **Prévoyez** chaque échec : une ligne ERROR dans le journal, puis `sys.exit` avec le bon code.
3. **Terminez** par `sys.exit(0)`.
4. **Testez** à la main, puis planifiez, avec le dossier de départ.
5. **Vérifiez** une vraie exécution : heure, code, journal.

Un exemple concret

Le Planificateur affiche 3 comme dernier résultat de l'inventaire du lundi.

Étape	Ce qu'on fait	Résultat
1	Lire l'heure de la dernière exécution	Lundi, 8 h 00 : la tâche a bien tourné.
2	Lire le code : 3	Un échec prévu, pas un plantage.
3	Lire le journal à 8 h 00	<code>ERROR Dossier introuvable : partage</code>
4	Chercher la cause	Le dossier de départ manque dans l'action.

Conclusion : on corrige l'action de la tâche, on la relance à la main, et on vérifie que le code revient à 0.

Les pièges

- **✗ Afficher « Erreur » sans `sys.exit`** → le code vaut 0, personne n'est prévenu.
- **✗ Oublier le dossier de départ** → le script cherche ses fichiers ailleurs.
- **✗ Croire une tâche créée** → vérifier une exécution réelle.
- **✗ Sauvegarder sans jamais restaurer** → restaurer dans une base neuve et comparer.

Script d'inventaire, journal et restauration

LABO-U6-S1-07

Contexte — Marc Oliveira, le trésorier de MaisonDesJeux, veut deux choses. Savoir chaque semaine ce que contient le dossier partagé de l'association. Et ne plus jamais perdre le fichier des adhérents.

Bruno Sanchez, bénévole, a écrit un début de script d'inventaire. Il marche, mais seulement sur un dossier écrit dans le code, et il ne dit jamais s'il a échoué. Vous le rendez prêt à tourner seul, vous le planifiez, puis vous prouvez qu'on sait restaurer la base.

RÈGLE IA DE LA SÉANCE

Scénario 1 : pas d'IA pendant ce labo. Vous avez les annexes, l'aide `--help` et la recherche web classique.

Bloqué ? Relisez l'annexe citée dans l'étape, puis demandez au formateur.

À la fin de ce labo, vous saurez régler un script par ses arguments. Il écrira un journal et rendra un code de retour. Vous saurez aussi le planifier et prouver une restauration.

Comment ça se valide — à chaque point de contrôle, vous montrez votre résultat au formateur. Il valide, puis vous passez à la suite.

Les mots de la séance

Mot	Ce que ça veut dire
Script	Un petit programme sans fenêtre, qui fait une tâche à la place d'un humain.
Argument	Une valeur donnée au script sur la ligne de commande : <code>--dossier partage</code> .
Code de retour	Le nombre rendu par le script quand il s'arrête. 0 veut dire « réussi ».
Journal	Le fichier texte où le script note ce qu'il fait, avec la date et l'heure.
Planificateur de tâches	Le programme de Windows qui lance une tâche à l'heure prévue.
Inventaire	La liste de tous les fichiers d'un dossier, avec leur taille.

Mot	Ce que ça veut dire
Empreinte	Un code court calculé sur tout le contenu d'une table. Un seul changement, et elle change.

Ce dont vous avez besoin

- Votre poste Windows, avec Python, VS Code et PowerShell.
- [Annexe A1](#) : le dossier de travail et son mode d'emploi. Il se télécharge depuis l'[annexe A1](#), dans le fichier [A1_mdj_scripts.zip](#).
- [Annexe A2](#) : un extrait de la documentation de Python.
- [Annexe A3](#) : les règles d'exploitation des scripts de MaisonDesJeux.
- [Annexe A4](#) : un extrait de la documentation du Planificateur de tâches.

En binôme — le pilote tape, le lecteur lit les consignes et les annexes à voix haute, et vérifie chaque résultat. On inverse à la partie C.

Le plan du labo

Partie	Étapes	Ce que vous obtenez	Temps
A — Un script réglé par ses arguments	1 à 6	l'inventaire de n'importe quel dossier, sans toucher au code	30 min
B — Un journal et des codes de retour	7 à 12	un script qui dit s'il a réussi, et pourquoi il a échoué	30 min
C — Planifier le script	13 à 18	un inventaire lancé par Windows, et son résultat lu	30 min
D — Sauvegarder et restaurer la base	19 à 24	une restauration prouvée par la comparaison	30 min

PARTIE A — Un script réglé par ses arguments

Étapes 1 à 6, environ 30 min. À la fin, le script fait l'inventaire du dossier que vous lui donnez sur la ligne de commande.

Besoin d'un rappel ? Revoir le cours : « [Un script qu'on règle sans toucher au code](#) ».

01 Installer le dossier de travail

- ☐ Téléchargez `A1_mdj_scripts.zip` depuis l'[annexe A1](#).
- ☐ Faites un clic droit sur le fichier, puis « Extraire tout ». Avec « Parcourir », choisissez votre dossier « Documents ».

Vous devez voir : un dossier `mdj-scripts` dans « Documents ». Il contient `adherents.db`, `inventaire.py`, `restaurer.py`, `verifier_base.py` et le dossier `partage`.

Si vous voyez un dossier `mdj-scripts` dans un autre dossier `mdj-scripts` : utilisez celui qui contient directement `inventaire.py`.

02 Ouvrir le dossier dans VS Code

- ☐ Ouvrez VS Code. Menu « Fichier », puis « Ouvrir le dossier ». Choisissez `Documents\mdj-scripts`.
- ☐ Ouvrez un terminal : menu « Terminal », puis « Nouveau terminal ». Tapez :

```
ls
```

Vous devez voir : cinq éléments, dont `inventaire.py` et `partage`. Le terminal est déjà dans le dossier `mdj-scripts` : c'est votre dossier courant.

03 Lancer la version de départ

☐ Ouvrez `inventaire.py` et lisez-le. Son code est aussi dans l'[annexe A1](#).

☐ Dans le terminal, tapez :

```
python inventaire.py
```

Vous devez voir : deux lignes. L'heure dans le nom du rapport sera la vôtre.

```
20 fichiers, 51600 octets  
Rapport : rapports\inventaire_2026-10-07_140512.csv
```

Si vous voyez « python n'est pas reconnu » : tapez `py` à la place de `python`, ici et dans toute la suite.

Pourquoi ? Marc veut aussi l'inventaire du seul dossier `partage\public`. Avec cette version, que faut-il faire à chaque fois ?

Votre réponse :

04 Trouver dans la doc

Vous allez remplacer le dossier écrit dans le code par un argument. Le module qui lit les arguments s'appelle `argparse`.

☐ Ouvrez l'[annexe A2](#), partie « Lire les arguments : `argparse` ».

☐ Trouvez le réglage qui rend un argument obligatoire. Notez-le ici :

☐ Trouvez le réglage qui donne une valeur quand l'argument est absent. Notez-le ici :

Vous devez avoir : deux mots écrits entre les parenthèses de `add_argument` . Vous les utilisez à l'étape suivante.

05 Ajouter les arguments

☐ Tout en haut de `inventaire.py` , ajoutez la ligne `import argparse` , au-dessus de `import csv` .

☐ Remplacez les deux lignes `dossier = Path("partage")` et `sortie = Path("rapports")` par ce bloc :

```
parser = argparse.ArgumentParser(description="Inventaire d'un dossier.")
parser.add_argument("--dossier", required=True, help="le dossier à inventorier")
parser.add_argument("--sortie", default="rapports", help="le dossier des rapports")
args = parser.parse_args()

dossier = Path(args.dossier)
sortie = Path(args.sortie)
```

☐ Enregistrez avec Ctrl+S.

Vous devez voir : aucun point rouge sur l'onglet de `inventaire.py` dans VS Code. Le fichier est enregistré.

06 Tester les arguments

☐ Tapez, une commande après l'autre :

```
python inventaire.py --dossier partage
python inventaire.py --dossier partage\animations
python inventaire.py --help
```

Vous devez voir : 20 fichiers et 51600 octets pour la première commande. 9 fichiers et 24300 octets pour la deuxième. Pour la troisième, le mode d'emploi :

```
usage: inventaire.py [-h] --dossier DOSSIER [--sortie SORTIE]

Inventaire d'un dossier.

options:
  -h, --help            show this help message and exit
  --dossier DOSSIER      le dossier à inventorier
  --sortie SORTIE        le dossier des rapports
```

Pourquoi ? Dans le mode d'emploi, `--sortie SORTIE` est entre crochets, mais pas `--dossier DOSSIER`. Que veulent dire ces crochets ?

Votre réponse :

POINT DE CONTRÔLE A

- ☐ Montrez au formateur l'inventaire de `partage\animations`, lancé avec l'argument `--dossier`.
- ☐ Le formateur a validé. Passez à la partie B.

PARTIE B — Un journal et des codes de retour

Étapes 7 à 12, environ 30 min. À la fin, le script rend un code qui dit s'il a réussi, et il note chaque lancement dans son journal.

Besoin d'un rappel ? Revoir le cours : [« Dire si ça a marché : code de retour et journal »](#).

07 Lire le code de retour

Après chaque commande, PowerShell range le code de retour dans la variable `$LASTEXITCODE`.

- ☐ Tapez ces deux lignes :

```
python inventaire.py --dossier partage
$LASTEXITCODE
```

Vous devez voir : l'inventaire, puis `0`.

☐ Lancez le script sans argument, puis lisez le code :

```
python inventaire.py
$LASTEXITCODE
```

Vous devez voir :

```
usage: inventaire.py [-h] --dossier DOSSIER [--sortie SORTIE]
inventaire.py: error: the following arguments are required: --dossier
2
```

08 Découvrir un échec silencieux

☐ Lancez l'inventaire d'un dossier qui n'existe pas. Attention au « s » en trop :

```
python inventaire.py --dossier partages
$LASTEXITCODE
```

Vous devez voir : `0` fichiers, `0` octets, un rapport vide, puis `0`.

Le dossier n'existe pas, et pourtant le script dit « réussi ». Un Planificateur ne verrait aucun problème.

09 Lire les règles de l'équipe

☐ Ouvrez l'[annexe A3](#), partie « Les codes de retour ».

- ☐ Notez le code prévu quand le dossier à traiter est introuvable :

Vous devez voir : un tableau de six codes, de 0 à 5, avec pour chacun son sens et qui le rend.

Pourquoi ? Pourquoi l'équipe ne choisit-elle pas les codes 1 et 2 pour ses propres échecs ?

Votre réponse :

10 Rendre le bon code

- ☐ Tout en haut du fichier, ajoutez `import sys`.
- ☐ Juste sous la ligne `sortie = Path(args.sortie)`, ajoutez ce bloc :

```
if not dossier.is_dir():
    print("Erreur : dossier introuvable :", dossier)
    sys.exit(3)
```

- ☐ Tout à la fin du fichier, ajoutez une dernière ligne, sans retrait : `sys.exit(0)`.
- ☐ Enregistrez, puis relancez :

```
python inventaire.py --dossier partages
$LASTEXITCODE
```

Vous devez voir : Erreur : dossier introuvable : partages , puis 3 . Aucun rapport n'est créé.

11 Ajouter le journal

☐ Tout en haut du fichier, ajoutez `import logging` .

☐ Juste sous la ligne `args = parser.parse_args()` , ajoutez ce bloc :

```
logging.basicConfig(filename="inventaire.log", level=logging.INFO,
                    format="%(asctime)s %(levelname)s %(message)s",
                    encoding="utf-8")
logging.info("Début de l'inventaire de %s", args.dossier)
```

☐ Dans le bloc `if not dossier.is_dir():` , ajoutez cette ligne juste avant le `print` :

```
logging.error("Dossier introuvable : %s", dossier)
```

☐ Juste avant la ligne `print(len(fichiers), ...)` , ajoutez :

```
logging.info("Inventaire terminé : %d fichiers, %d octets", len(fichiers), total)
```

☐ Enregistrez.

Vous devez voir : le code complet de cette étape est dans l'[annexe A1](#), partie « Le script à la fin de l'étape 11 ». Comparez le vôtre avec lui, ligne par ligne.

12 Remplir la grille des trois cas

☐

Lancez ces trois commandes. Après chacune, tapez `$LASTEXITCODE` et notez le code :

```
python inventaire.py --dossier partage
python inventaire.py
python inventaire.py --dossier partages
```

☐ Affichez le journal. L'option `-Encoding UTF8` garde les accents :

```
Get-Content inventaire.log -Encoding UTF8
```

☐ Remplissez la grille. La première ligne est faite pour vous montrer.

Cas	Code de retour	Dernière ligne ajoutée au journal, ou « rien »
<code>--dossier partage</code>	0	<code>INFO Inventaire terminé : 20 fichiers, 51600 octets</code>

Cas	Code de retour	Dernière ligne ajoutée au journal, ou « rien »
-----	----------------	--

sans argument

`--dossier partages`

Vous devez voir : trois codes différents dans votre grille, et une seule ligne ERROR dans le journal.

Pourquoi ? Pour le lancement sans argument, aucune ligne n'est ajoutée au journal.

Regardez l'ordre des lignes du script. Pourquoi ?

Votre réponse :

POINT DE CONTRÔLE B

- ☐ Montrez au formateur votre grille remplie et votre fichier `inventaire.log`.
- ☐ Le formateur a validé. Passez à la partie C. Inversez les rôles : le lecteur devient pilote.

PARTIE C — Planifier le script

Étapes 13 à 18, environ 30 min. À la fin, Windows lance l'inventaire tout seul, et vous lisez le résultat de chaque exécution.

Besoin d'un rappel ? Revoir le cours : « [Lancer le script à heure fixe](#) ».

13 Trouver dans la doc

- ☐ Ouvrez l'[annexe A4](#), partie « Lire le résultat d'une tâche ».
- ☐ Notez la commande qui affiche le résultat de la dernière exécution :

- ☐ Notez le nom de la colonne qui contient le code de retour :

Vous devez avoir : une commande qui commence par `Get-`, et un nom de colonne en anglais.

14 Préparer l'action

L'action dit quoi lancer : le programme, ses arguments, et le dossier de départ.

-
- ☐ Dans le terminal de VS Code, toujours dans `mdj-scripts`, tapez ces trois lignes :
-

```
$dossier = (Get-Location).Path
$python = (Get-Command python).Source
$action = New-ScheduledTaskAction -Execute $python -Argument "inventaire.py --
dossier partage" -WorkingDirectory $dossier
```

-
- ☐ Vérifiez les deux chemins en tapant `$dossier`, puis `$python`.
-

Vous devez voir : le chemin de votre dossier `mdj-scripts`, puis un chemin qui finit par `python.exe`.

Si vous voyez une erreur rouge sur `Get-Command python` : remplacez `python` par `py` dans la deuxième ligne, puis retapez la troisième.

Ces variables vivent dans ce terminal. Si vous le fermez, retapez les trois lignes.

15 Créer la tâche planifiée

Le déclencheur dit quand lancer. Ici, dans une minute, puis toutes les deux minutes pendant deux heures.

-
- ☐ Tapez ces deux lignes :
-

```
$declencheur = New-ScheduledTaskTrigger -Once -At (Get-Date).AddMinutes(1)
-RepetitionInterval (New-TimeSpan -Minutes 2) -RepetitionDuration (New-TimeSpan
-Hours 2)
Register-ScheduledTask -TaskName "MDJ-inventaire" -Action $action -Trigger
$declencheur
```

Vous devez voir : un petit tableau avec `MDJ-inventaire` dans la colonne TaskName et `Ready` dans la colonne State.

Si vous voyez « Accès refusé » : le poste bloque la création de tâches. Appelez le formateur.

16 Lancer la tâche et lire son résultat

☐ Lancez la tâche tout de suite, sans attendre le déclencheur :

```
Start-ScheduledTask -TaskName "MDJ-inventaire"
```

☐ Attendez dix secondes. Puis tapez :

```
Get-ScheduledTaskInfo -TaskName "MDJ-inventaire" | Select-Object LastRunTime,
LastTaskResult, NextRunTime
```

Vous devez voir : l'heure de l'exécution dans `LastRunTime`, et `0` dans `LastTaskResult`.

Si vous voyez 267009 : la tâche tourne encore. Attendez dix secondes et recommencez. `267011` veut dire qu'elle n'a pas encore tourné.

☐ Affichez les dernières lignes du journal :

```
Get-Content inventaire.log -Encoding UTF8 -Tail 2
```

Vous devez voir : deux lignes INFO, à l'heure de `LastRunTime`.

Pourquoi ? Le Planificateur dit déjà `0`. Pourquoi lire aussi le journal ?

Votre réponse :

17 Voir une exécution automatique

☐ Notez l'heure de `NextRunTime` : _____

☐ Quand cette heure est passée, retapez la commande `Get-ScheduledTaskInfo` de l'étape 16.

Vous devez voir : `LastRunTime` a changé et vaut l'heure notée. `LastTaskResult` vaut `0`.
Personne n'a lancé le script : c'est Windows.

☐ Comptez les rapports produits :

```
(Get-ChildItem rapports).Count
```

Vous devez voir : un rapport de plus qu'à l'étape 16.

18 Casser la tâche, puis la supprimer

- ☐ Changez l'action : la tâche va chercher un dossier qui n'existe pas.

```
$action2 = New-ScheduledTaskAction -Execute $python -Argument "inventaire.py --  
dossier partages" -WorkingDirectory $dossier  
Set-ScheduledTask -TaskName "MDJ-inventaire" -Action $action2  
Start-ScheduledTask -TaskName "MDJ-inventaire"
```

- ☐ Attendez dix secondes, puis relisez le résultat avec la commande de l'étape 16.

Vous devez voir : 3 dans `LastTaskResult`. Le journal se termine par `ERROR Dossier introuvable : partages`.

- ☐ Supprimez la tâche :

```
Unregister-ScheduledTask -TaskName "MDJ-inventaire" -Confirm:$false
```

Vous devez voir : aucun message. La commande `Get-ScheduledTask -TaskName "MDJ-inventaire"` affiche maintenant une erreur : la tâche n'existe plus.

POINT DE CONTRÔLE C

- ☐ Montrez au formateur les lignes du journal écrites par la tâche, avec leurs heures, et la ligne ERROR de l'étape 18.
- ☐ Le formateur a validé. Passez à la partie D.

PARTIE D — Sauvegarder et restaurer la base

Étapes 19 à 24, environ 30 min. À la fin, vous avez prouvé qu'une sauvegarde de la base des adhérents se restaure à l'identique.

Besoin d'un rappel ? Revoir le cours : [« Sauvegarder, et prouver qu'on sait restaurer »](#).

19 Noter l'état de la base

L'outil `verifier_base.py` compte les lignes de chaque table et calcule leur empreinte. Son mode d'emploi est dans l'[annexe A1](#).

☐

Tapez :

```
python verifier_base.py adherents.db
```

Vous devez voir : Intégrité : ok , puis une ligne par table : `adherent` , `jeu` , `emprunt` .

☐

Recopiez les nombres de lignes et les empreintes dans la colonne « Avant » de la grille de l'étape 24.

20 Écrire le script de sauvegarde

☐

Dans VS Code, créez un fichier `sauvegarde.py` dans `mdj-scripts` . Écrivez :

```

"""Sauvegarde de la base des adhérents dans un fichier .sql daté."""
import sqlite3
from datetime import date

con = sqlite3.connect("adherents.db")
nom = "sauvegarde_" + date.today().isoformat() + ".sql"
with open(nom, "w", encoding="utf-8") as f:
    for ligne in con.iterdump():
        f.write(ligne + "\n")
con.close()
print("Sauvegarde écrite :", nom)

```

- ☐ Enregistrez, puis lancez `python sauvegarde.py`.

Vous devez voir : Sauvegarde écrite : sauvegarde_ suivi de la date du jour et de .sql.

- ☐ Affichez les trois premières lignes du fichier :

```
Get-Content sauvegarde_*.sql -Encoding UTF8 -TotalCount 3
```

Vous devez voir : BEGIN TRANSACTION; , puis CREATE TABLE adherent (. La sauvegarde est une suite de commandes SQL qui reconstruisent la base.

21 Ranger la copie ailleurs

- ☐ Créez un dossier de sauvegardes, à côté de `mdj-scripts`, puis déplacez-y la copie :

```

New-Item -ItemType Directory ..\sauvegardes-mdj
Move-Item sauvegarde_*.sql ..\sauvegardes-mdj

```

Vous devez voir : `ls ..\sauvegardes-mdj` affiche votre fichier `sauvegarde_....sql`.

Pourquoi ? Dans une vraie installation, où faudrait-il ranger ce dossier ? Et pourquoi pas sur le même disque que la base ?

Votre réponse :

22 Simuler la perte de la base

☐ Supprimez la base :

```
Remove-Item adherents.db
```

☐ Vérifiez qu'elle a disparu :

```
python verifier_base.py adherents.db  
$LASTEXITCODE
```

Vous devez voir : Base introuvable : adherents.db , puis 3 .

23 Restaurer la base

L'outil `restaurer.py` reconstruit une base à partir d'une sauvegarde. Il refuse de remplacer une base qui existe.

☐ Mettez le chemin de la copie dans une variable, puis restaurez :

```
$copie = Get-ChildItem ..\sauvegardes-mdj\sauvegarde_*.sql | Select-Object -First 1
python restaurer.py $copie.FullName adherents.db
```

Vous devez voir : Base restaurée : adherents.db.

24 Prouver la restauration

☐

Relancez `python verifier_base.py adherents.db`.

☐

Remplissez la colonne « Après ». Comparez chaque ligne avec « Avant ».

Table	Avant : lignes et empreinte	Après : lignes et empreinte	Identique ?
adherent			
jeu			
emprunt			

Vous devez voir : les trois lignes identiques. Si une seule diffère, la restauration n'est pas prouvée.

Pourquoi ? Pourquoi comparer les empreintes, et pas seulement vérifier que `adherents.db` existe de nouveau ?

Votre réponse :

POINT DE CONTRÔLE D

- ☐ Montrez au formateur votre grille « Avant » et « Après », avec les empreintes identiques.
- ☐ Le formateur a validé. Le labo est terminé.

POUR ALLER PLUS LOIN

15 min, si vous avez fini avant la fin de la séance.

1. Le code 5 : droits insuffisants. L'[annexe A3](#) prévoit le code 5 quand le script ne peut pas écrire.

- ☐ Dans `inventaire.py`, entourez la création du dossier et l'écriture du rapport avec `try:` et `except PermissionError:`. Dans le `except`, écrivez une ligne `ERROR` dans le journal, puis `sys.exit(5)`.
- ☐ Interdisez-vous l'écriture dans `rapports`, puis lancez l'inventaire :

```
icacIs rapports /deny "${env:USERNAME}:(W)"
python inventaire.py --dossier partage
$LASTEXITCODE
```

Vous devez voir : 5, et une ligne `ERROR` dans le journal.

- ☐ Rendez-vous le droit d'écrire :

```
icacIs rapports /remove:d "${env:USERNAME}"
```

2. Le piège de la base absente. Créez un dossier vide `essai`, copiez-y `sauvegarde.py`, et lancez-le depuis ce dossier.

☐ Regardez ce que contient `essai` après le lancement, puis lancez `python ..\verifier_base.py adherents.db` dans `essai`.

☐ Notez ce que vous constatez : _____

CE QU'IL FAUT RETENIR

1. Un script se règle par ses arguments. Il se termine par `sys.exit` avec un code : `0` si tout va bien, le code prévu par l'équipe sinon.
2. Une tâche planifiée se vérifie sur une vraie exécution : `LastTaskResult` et la ligne du journal à la même heure.
3. Une restauration se prouve en comparant les lignes et les empreintes, avant et après, dans une base reconstruite.

Tous les documents du module

A1 — Le dossier de travail du labo et son mode d'emploi

↓ Télécharger A1_mdj_scripts.zip

copie : U6-S1-M7, [annexe A1](#)

Source : fichiers écrits pour le module. Schéma de la base : document de scénario, § Base de données. Fichier [A1_mdj_scripts.zip](#) à télécharger ci-dessus. Utilisée dans : LABO étapes 1 à 24.

Ce que contient le dossier

```
mdj-scripts/  
  adherents.db      la base des adhérents de MaisonDesJeux  
  inventaire.py     le script d'inventaire de Bruno : c'est lui que vous  
modifiez  
  verifier_base.py  l'outil qui vérifie une base : fourni  
  restaurer.py      l'outil qui restaure une base depuis une sauvegarde .sql :  
fourni  
  partage/          une copie du dossier partagé de l'association, à  
inventorier  
  administration/  
  animations/  
  public/
```

Le dossier **partage** contient 20 fichiers répartis dans trois sous-dossiers.

La base adherents.db

Une base SQLite de trois tables.

Table	Colonnes	Lignes
adherent	id, nom, prenom, date_naissance, date_adhesion, tarif	340
jeu	id, titre, editeur, annee_edition	800
emprunt	id, id_adherent, id_jeu, date_emprunt, date_retour	1250

Les dates s'écrivent `AAAA-MM-JJ`. `date_retour` est vide tant que le jeu n'est pas rendu.

Le script `inventaire.py` de départ

```
"""Inventaire du partage de MaisonDesJeux.

Version de départ : le dossier à inventorier est écrit dans le code.
Le script écrit un rapport CSV dans le dossier rapports.
"""

import csv
from datetime import datetime
from pathlib import Path

dossier = Path("partage")
sortie = Path("rapports")

fichiers = [f for f in dossier.rglob("*") if f.is_file()]
total = sum(f.stat().st_size for f in fichiers)

sortie.mkdir(exist_ok=True)
rapport = sortie / ("inventaire_" + datetime.now().strftime("%Y-%m-%d_%H%M%S") + ".csv")
with open(rapport, "w", newline="", encoding="utf-8") as f:
    ecrivain = csv.writer(f, delimiter=";")
    ecrivain.writerow(["fichier", "taille_octets"])
    for fichier in fichiers:
        ecrivain.writerow([fichier.relative_to(dossier), fichier.stat().st_size])

print(len(fichiers), "fichiers,", total, "octets")
print("Rapport :", rapport)
```

`dossier.rglob("*")` parcourt le dossier et tous ses sous-dossiers. `f.stat().st_size` donne la taille d'un fichier, en octets.

Le script à la fin de l'étape 11

Votre fichier peut avoir les imports dans un autre ordre : c'est juste.

```

"""Inventaire du partage de MaisonDesJeux.

Version de départ : le dossier à inventorier est écrit dans le code.
Le script écrit un rapport CSV dans le dossier rapports.
"""

import argparse
import csv
import logging
import sys
from datetime import datetime
from pathlib import Path

parser = argparse.ArgumentParser(description="Inventaire d'un dossier.")
parser.add_argument("--dossier", required=True, help="le dossier à inventorier")
parser.add_argument("--sortie", default="rapports", help="le dossier des rapports")
args = parser.parse_args()
logging.basicConfig(filename="inventaire.log", level=logging.INFO,
                    format="%(asctime)s %(levelname)s %(message)s",
                    encoding="utf-8")
logging.info("Début de l'inventaire de %s", args.dossier)

dossier = Path(args.dossier)
sortie = Path(args.sortie)
if not dossier.is_dir():
    logging.error("Dossier introuvable : %s", dossier)
    print("Erreur : dossier introuvable :", dossier)
    sys.exit(3)

fichiers = [f for f in dossier.rglob("*") if f.is_file()]
total = sum(f.stat().st_size for f in fichiers)

sortie.mkdir(exist_ok=True)
rapport = sortie / ("inventaire_" + datetime.now().strftime("%Y-%m-%d_%H%M%S") +
".csv")
with open(rapport, "w", newline="", encoding="utf-8") as f:
    ecrivain = csv.writer(f, delimiter=";")
    ecrivain.writerow(["fichier", "taille_octets"])
    for fichier in fichiers:
        ecrivain.writerow([fichier.relative_to(dossier), fichier.stat().st_size])

logging.info("Inventaire terminé : %d fichiers, %d octets", len(fichiers), total)
print(len(fichiers), "fichiers,", total, "octets")
print("Rapport :", rapport)
sys.exit(0)

```

L'outil verifier_base.py

Il vérifie une base sans la modifier : il l'ouvre en lecture seule.

```
python verifier_base.py adherents.db
```

Il affiche l'intégrité de la base, puis, pour chaque table, le nombre de lignes et l'empreinte.
Exemple de sortie :

```
Base : adherents.db
Intégrité : ok
adherent : 340 lignes, empreinte 1693833f
jeu      : 800 lignes, empreinte 7bcb4f34
emprunt  : 1250 lignes, empreinte 7bdb0d05
```

L'empreinte est un code de huit caractères, calculé sur tout le contenu de la table. Deux tables qui ont la même empreinte ont exactement le même contenu.

Code de retour	Sens
0	la base est lisible et intègre
1	la base est abîmée
2	mauvais usage : il manque le nom de la base
3	la base est introuvable

L'outil restaurer.py

Il reconstruit une base à partir d'une sauvegarde `.sql`. Il ne remplace jamais une base qui existe déjà.

```
python restaurer.py SAUVEGARDE.sql BASE.db
```

Exemple de sortie : `Base restaurée : adherents.db`.

Code de retour	Sens
0	la base est restaurée
1	la base existe déjà : rien n'est fait
2	mauvais usage : il manque un des deux noms
3	la sauvegarde est introuvable

A2 — Extrait de la documentation de Python

copie : U6-S1-M7, [annexe A2](#)

Source : documentation officielle de Python 3, docs.python.org, traduite et raccourcie pour le module. Utilisée dans : LABO étape 4.

Lire les arguments : `argparse`

```
import argparse
parser = argparse.ArgumentParser(description="Ce que fait le script.")
parser.add_argument("--nom", ...)
args = parser.parse_args()
```

Réglages de `add_argument` :

Réglage	Effet
<code>required=True</code>	L'argument est obligatoire. S'il manque, le script s'arrête avec le code 2.
<code>default=valeur</code>	La valeur utilisée quand l'argument est absent.
<code>type=int</code>	Convertit la valeur en nombre entier. Sans lui, la valeur est un texte.
<code>help="texte"</code>	La phrase affichée par <code>--help</code> .

Après `parse_args()`, chaque valeur se lit avec `args.nom`. Un tiret dans le nom devient un tiret bas : `--espace-min` se lit `args.espace_min`.

Le réglage `-h` ou `--help` est ajouté tout seul : il affiche le mode d'emploi. Dans ce mode d'emploi, un argument entre crochets est facultatif.

Arrêter le script avec un code : `sys.exit`

```
import sys
sys.exit(3)
```

Le script s'arrête tout de suite et rend le code donné. `sys.exit(0)` veut dire « réussi ».

Sans `sys.exit`, un script qui va jusqu'au bout rend 0. Un script qui plante rend 1.

Écrire un journal : logging

```
import logging
logging.basicConfig(filename="script.log", level=logging.INFO,
                    format="%(asctime)s %(levelname)s %(message)s",
                    encoding="utf-8")
logging.info("Début du traitement")
logging.error("Fichier introuvable : %s", nom)
```

Réglage de	Effet
<code>basicConfig</code>	
<code>filename</code>	Le fichier journal. Les lignes s'ajoutent à la fin, le fichier n'est jamais vidé.
<code>level=logging.INFO</code>	Les lignes INFO, WARNING et ERROR sont écrites.
<code>format</code>	<code>%(asctime)s</code> écrit la date et l'heure, <code>%(levelname)s</code> le niveau, <code>%(message)s</code> le message.
<code>encoding="utf-8"</code>	L'encodage du fichier journal.

Dans le message, `%s` est remplacé par un texte et `%d` par un nombre entier, dans l'ordre des valeurs qui suivent.

Lire une base : sqlite3

```
import sqlite3
con = sqlite3.connect("base.db")
lignes = con.execute("SELECT nom, prenom FROM adherent ORDER BY id").fetchall()
con.close()
```

`fetchall()` renvoie la liste de toutes les lignes. Chaque ligne est un tuple de valeurs.

`con.iterdump()` donne, une à une, les commandes SQL qui reconstruisent toute la base.

Attention : si le fichier donné à `connect` n'existe pas, `sqlite3` ne signale aucune erreur. Il crée une base vide avec ce nom. Vérifiez que le fichier existe avant de l'ouvrir.

Mesurer l'espace libre : `shutil.disk_usage`

```
import shutil
espace = shutil.disk_usage("C:\\")
espace.free
```

`disk_usage` reçoit un chemin. Il renvoie la taille totale, la place utilisée et la place libre du disque, en octets : `espace.total`, `espace.used`, `espace.free`. Un méga-octet vaut `1024 * 1024` octets.

Si Python n'a pas le droit de lire le chemin donné, `disk_usage` lève une `PermissionError`.

Chemins : `pathlib`

Écrit dans le code	Effet
<code>Path("exports")</code>	un chemin, relatif au dossier courant
<code>p.is_file()</code>	vrai si <code>p</code> est un fichier qui existe
<code>p.is_dir()</code>	vrai si <code>p</code> est un dossier qui existe
<code>p.mkdir(exist_ok=True)</code>	crée le dossier, sans erreur s'il existe déjà
<code>p.resolve().anchor</code>	le disque du chemin, par exemple <code>C:\</code>
<code>dossier / "nom.csv"</code>	le chemin du fichier <code>nom.csv</code> dans <code>dossier</code>

Écrire un fichier : `open` et l'encodage

```
with open("export.csv", "w", newline="", encoding="utf-8-sig") as f:
    ...
```

Encodage	Quand l'utiliser
<code>"utf-8"</code>	Le cas général : journaux, scripts, fichiers lus par des programmes.
<code>"utf-8-sig"</code>	UTF-8 avec BOM. Le BOM est une marque invisible au début du fichier. Elle permet aux tableurs, comme Excel, d'afficher les accents correctement.

Si l'écriture est interdite, `open` lève une `PermissionError`.

La date du jour : datetime

```
from datetime import date
date.today().isoformat()      # "2026-10-01"
date.today().strftime("%Y-%m") # "2026-10"
```

`%Y` est l'année sur quatre chiffres, `%m` le mois sur deux chiffres, `%d` le jour.

A3 — Les règles d'exploitation des scripts de MaisonDesJeux

copie : U6-S1-M7, [annexe A3](#)

Source : règles écrites pour l'association par le prestataire, validées par Marc Oliveira. Utilisée dans : LABO étape 9.

Ces règles valent pour tous les scripts de l'association. Elles permettent à Bruno Sanchez, ou à un autre bénévole, de comprendre un script qu'il n'a pas écrit.

1. Les réglages

- Chaque réglage est un argument : un dossier, un fichier, un seuil. Jamais une valeur écrite dans le code.
- Chaque argument a une valeur par défaut quand c'est possible.
- Le script ne pose jamais de question avec `input()`.

2. Le journal

- Le journal porte un nom court, qui dit ce que fait le script, suivi de `.log` :
`inventaire.py` écrit dans `inventaire.log`, `export_mensuel.py` dans `export.log`.
- Il est dans le dossier de départ du script.
- Chaque ligne suit le format `date heure NIVEAU message`, en UTF-8.
- Une ligne INFO au début, une ligne INFO à la fin quand tout est fait.
- Une ligne ERROR à chaque échec, qui dit la cause.

3. Les codes de retour

Code	Sens	Qui le rend
0	tout s'est bien passé	le script, avec <code>sys.exit(0)</code>
1	le script a planté sur une erreur imprévue	Python, tout seul
2	les arguments sont faux ou manquent	argparse, tout seul
3	le fichier, la base ou le dossier à traiter est introuvable	le script
4	l'espace libre sur le disque est insuffisant	le script
5	le script n'a pas le droit d'écrire	le script

Un script ne rend jamais 0 après un échec.

4. Les données

- Un script ne remplace jamais une base qui existe.
- Une sauvegarde porte la date dans son nom, sous la forme `AAAA-MM-JJ`.
- Une sauvegarde est rangée hors du dossier de la base.

5. Les tâches planifiées

- Le nom d'une tâche commence par `MDJ-` : `MDJ-inventaire`, `MDJ-export-mensuel`.
- Le dossier de départ est toujours le dossier du script.
- Une tâche est testée à la main avant d'être laissée seule.

A4 — Extrait de la documentation du Planificateur de tâches

copie : U6-S1-M7, [annexe A4](#)

Source : documentation de Microsoft, module PowerShell ScheduledTasks, traduite et raccourcie pour le module. Utilisée dans : LABO étapes 13 à 18.

Ces commandes marchent dans PowerShell, sans droits administrateur, pour les tâches de votre propre compte. Par défaut, une tâche ne se lance que si votre session Windows est ouverte.

Créer une tâche

Une tâche se crée en trois commandes : l'action, le déclencheur, puis l'enregistrement.

L'action : quoi lancer.

```
$action = New-ScheduledTaskAction -Execute CHEMIN_DU_PROGRAMME -Argument "ARGUMENTS"
-WorkingDirectory DOSSIER_DE_DÉPART
```

Paramètre	Ce qu'il reçoit
<code>-Execute</code>	le chemin complet du programme, par exemple celui de <code>python.exe</code> . <code>(Get-Command python).Source</code> le donne.
<code>-Argument</code>	tout ce qui suit le programme : le script, puis ses arguments.
<code>-WorkingDirectory</code>	le dossier de départ. Les chemins relatifs du script partent de là.

Le déclencheur : quand lancer.

```
New-ScheduledTaskTrigger -Once -At (Get-Date).AddMinutes(1) -RepetitionInterval
(New-TimeSpan -Minutes 2) -RepetitionDuration (New-TimeSpan -Hours 1)
New-ScheduledTaskTrigger -Daily -At 6am
New-ScheduledTaskTrigger -Weekly -DaysOfWeek Monday -At 8am
```

La première ligne lance la tâche dans une minute, puis toutes les deux minutes pendant une heure : c'est le réglage des tests.

La deuxième, chaque jour à 6 h. La troisième, chaque lundi à 8 h. Cette commande ne sait pas faire un déclencheur mensuel : voir plus bas.

L'enregistrement.

```
Register-ScheduledTask -TaskName "NOM" -Action $action -Trigger $declencheur
```

Lancer, modifier, supprimer

Commande	Effet
<code>Start-ScheduledTask -TaskName "NOM"</code>	lance la tâche tout de suite, sans attendre le déclencheur

Commande	Effet
<code>Set-ScheduledTask -TaskName "NOM" -Action \$action</code>	remplace l'action de la tâche
<code>Set-ScheduledTask -TaskName "NOM" -Trigger \$declencheur</code>	remplace le déclencheur
<code>Get-ScheduledTask -TaskName "NOM"</code>	affiche la tâche et son état : <code>Ready</code> prête, <code>Running</code> en cours
<code>Unregister-ScheduledTask -TaskName "NOM" -Confirm:\$false</code>	supprime la tâche, sans demander de confirmation

Lire le résultat d'une tâche

```
Get-ScheduledTaskInfo -TaskName "NOM" | Select-Object LastRunTime, LastTaskResult, NextRunTime
```

Colonne	Contenu
<code>LastRunTime</code>	la date et l'heure de la dernière exécution
<code>LastTaskResult</code>	le code de retour rendu par le programme à la dernière exécution
<code>NextRunTime</code>	la date et l'heure de la prochaine exécution prévue

Valeur de <code>LastTaskResult</code>	Sens
<code>0</code>	le programme a rendu 0
<code>1, 2, 3 ...</code>	le programme a rendu ce code
<code>267009</code>	la tâche est en cours d'exécution
<code>267011</code>	la tâche ne s'est encore jamais lancée

Passer au déclencheur mensuel

Le déclencheur « le 1er de chaque mois » se règle dans la fenêtre du Planificateur de tâches.

1. Appuyez sur Windows + R, tapez `taskschd.msc`, puis Entrée.
2. À gauche, cliquez sur « Bibliothèque du Planificateur de tâches ».

3. Au centre, double-cliquez sur votre tâche.
4. Ouvrez l'onglet « Déclencheurs ». Sélectionnez le déclencheur, puis cliquez sur « Modifier ».
5. Choisissez « Mensuel ». Réglez l'heure de début sur 06:00:00.
6. Dans « Mois », choisissez « Sélectionner tous les mois ». Dans « Jours », cochez 1.
7. Décochez « Répéter la tâche toutes les ». Cliquez sur OK, puis encore sur OK.

L'onglet « Déclencheurs » affiche alors : « Mensuel », à 06:00, le 1er de chaque mois.

Pour comparer : cron sous Linux

Sur un serveur Linux, la même tâche mensuelle s'écrit en une ligne dans la table cron :

```
0 6 1 * * cd /srv/export-mensuel && python3 export_mensuel.py
```

Les cinq premières valeurs disent : minute 0, heure 6, jour 1, tous les mois, tous les jours de la semaine.