Fiche de TP/TD 3

(MSP2)

Informatique 3 : Programmation en Python

Gestion de fichiers, erreurs et exceptions

Exercice 1

(Numérotation de lignes d'un fichier texte)
Écrire un programme qui ouvre un fichier texte `"texte.txt"` en lecture et affiche chaque ligne précédée de son numéro (en commençant à 1).
Exemple de sortie : `1: Première ligne`, `2: Deuxième ligne`, etc.
Utiliser le mot-clé `with` pour garantir la fermeture du fichier.

Exercice 2

(Journal interactif dans un fichier)
Écrire un programme qui demande à l'utilisateur de saisir des phrases et les enregistre dans un fichier `"journal.txt"` en mode ajout (`"a"`).
L'utilisateur peut taper une ligne vide pour terminer.
Afficher un message confirmant la sauvegarde des lignes saisies.

Exercice 3

(Moyenne des notes à partir d'un fichier)
On dispose d'un fichier `"notes.txt"` contenant une note par ligne (format : un nombre réel).
Écrire un programme qui lit le fichier, calcule la moyenne des notes et l'affiche.
Gérer les erreurs possibles : fichier absent, ligne vide, valeur non numérique (dans ce cas, ignorer la ligne et afficher un avertissement).

Exercice 4

(Filtrage de lignes par mot-clé)
Écrire un programme qui lit un fichier `"log.txt"` et écrit dans un fichier `"erreurs.txt"` uniquement les lignes contenant le mot `"ERROR"`.
Utiliser une lecture ligne par ligne (boucle `for ligne in f`) et la méthode `in` sur les chaînes.
Afficher le nombre de lignes copiées.

Exercice 5

(Création de dossier et fichier avec `os`)
Écrire un programme qui :
1) Affiche le répertoire courant avec `os.getcwd()`.
2) Vérifie l'existence d'un dossier `"donnees"` ; s'il n'existe pas, le crée avec `os.mkdir()`.
3) Change le répertoire courant vers `"donnees"` et crée un fichier `"info.txt"` contenant le chemin complet du nouveau répertoire.

Exercice 6

(Compteur lignes / mots / caractères)
Écrire un programme qui lit un fichier `"texte.txt"` et calcule :
— le nombre de lignes ;
— le nombre total de mots (en utilisant `split()`) ;
— le nombre total de caractères (y compris espaces et retours à la ligne).
Afficher ces trois valeurs de façon lisible.

Exercice 7

(Afficher les N premières lignes d'un fichier)
Demander à l'utilisateur :
— le nom d'un fichier ;
— un entier $N > 0$.
Écrire un programme qui affiche les N premières lignes du fichier, ou toutes les lignes s'il y en a moins.
Gérer au minimum l'exception `FileNotFoundError` et afficher un message explicite si le fichier n'existe pas.

Exercice 8

(Copie simple de fichier texte)
Demander à l'utilisateur le nom d'un fichier source et d'un fichier destination.
Écrire un programme qui copie le contenu du fichier source dans le fichier destination (en écrasant son contenu).
Gérer les erreurs de type `FileNotFoundError` pour le fichier source et afficher un message adapté.

Exercice 9

(Fusion de deux fichiers texte)
On dispose de deux fichiers texte `"partie1.txt"` et `"partie2.txt"`.
Écrire un programme qui crée un fichier `"fusion.txt"` contenant d'abord le contenu de `"partie1.txt"`, puis celui de `"partie2.txt"`.
Utiliser l'instruction `with open(...)` pour chaque fichier et gérer le cas où un des deux fichiers est manquant.

Exercice 10

(Divisions à partir d'un fichier)
Créer un fichier `"divisions.txt"` où chaque ligne contient deux entiers séparés par un espace (par exemple `"10 2"`).
Écrire un programme qui lit chaque ligne, interprète les deux entiers a et b, et affiche `a / b = résultat`.
Gérer explicitement les exceptions `ZeroDivisionError` (division par zéro) et `ValueError` (ligne mal formatée), sans arrêter tout le programme.

Exercice 11

(Analyseur de log avec lignes mal formées)
On dispose d'un fichier `"app.log"` dans lequel chaque ligne commence par un niveau de gravité : `"INFO"`, `"WARN"` ou `"ERROR"`, suivi d'un message. Certaines lignes peuvent être mal formées.
Écrire un programme qui :

— lit le fichier ligne par ligne ;
— compte le nombre de lignes de chaque type ;
— ignore les lignes mal formées mais les enregistre dans un fichier `"logErreurs.txt"` avec un message explicatif.
Utiliser des blocs `try/except` au niveau du traitement de chaque ligne pour éviter l'arrêt complet du programme.

Exercice 12

(Calcul de moyenne robuste avec exception personnalisée)
Créer une exception personnalisée `NoteInvalideError` dérivant de `Exception`.
On dispose d'un fichier `"notesEtudiants.txt"` où chaque ligne est de la forme `"Nom Note"`. Certaines lignes peuvent contenir des notes hors de l'intervalle $[0, 20]$ ou mal formatées.
Écrire un programme qui :
— lit chaque ligne, extrait la note ;
— lève `NoteInvalideError` si la note n'est pas dans $[0, 20]$;
— intercepte cette exception pour ignorer la ligne fautive tout en la journalisant dans `"notesInvalides.txt"` ;
— calcule la moyenne sur les notes valides et l'affiche avec le nombre de lignes ignorées.

Exercice 13

(Parcours de répertoire et résumé de fichiers texte)
À partir d'un dossier `"donnees"` (qui peut contenir plusieurs fichiers `.txt`), écrire un programme qui :
— liste tous les fichiers se terminant par `".txt"` ;
— pour chacun, compte le nombre de lignes et de caractères ;
— écrit un rapport dans `"resume.txt"` listant, pour chaque fichier, son nom et ses statistiques.
Gérer les exceptions possibles : `FileNotFoundError` si le dossier n'existe pas, `PermissionError` si un fichier n'est pas lisible, etc.

Exercice 14

(Copie binaire robuste)
Écrire une fonction `copier_binaire(source, destination)` qui copie un fichier binaire (par exemple une image `.jpg`) en lisant par blocs (par exemple 4096 octets).
Utiliser `with open(..., "rb")` et `with open(..., "wb")` pour gérer les flux binaires.
Gérer les exceptions suivantes : `FileNotFoundError` pour le fichier source, `PermissionError` pour le fichier destination. Afficher des messages explicites et ne jamais laisser le programme se terminer sur une trace brute.

Exercice 15

(Sauvegarde « atomique » de configuration)
On souhaite écrire un fichier de configuration `"config.ini"`. Pour éviter de corrompre un fichier existant en cas d'erreur, on utilisera une sauvegarde « atomique » :
1) écrire le contenu dans un fichier temporaire `"config.ini.tmp"` ;
2) si l'écriture se passe bien, renommer `"config.ini.tmp"` en `"config.ini"`.
Écrire le programme complet avec gestion des exceptions d'entrée/sortie (`IOError`, `OSError`), en veillant à ne pas laisser de fichier temporaire incohérent en cas d'erreur (utiliser `try/except/finally`).

Exercice 16

(Chargement de configuration avec valeurs par défaut)
Écrire un programme qui tente de lire un fichier `"config.txt"` contenant des paires `clé=valeur` (une par ligne).
Si le fichier n'existe pas, créer `"config.txt"` avec des valeurs par défaut (par exemple `theme=clair`, `langue=fr`) et continuer avec ces valeurs.
Si une ligne est mal formée (pas de signe `=`), lever et intercepter une exception (par exemple `ValueError`) et ignorer la ligne après journalisation.
Afficher finalement la configuration en vigueur.

Exercice 17

(Fonction `tail` sécurisée)
Programmer une fonction `tail(nom_fichier, n)` qui affiche les n dernières lignes d'un fichier texte (comme la commande `tail` sous Unix).
Contraintes :
— ne pas charger tout le fichier en mémoire si possible (lecture progressive) ;
— gérer les cas d'erreurs : fichier absent, $n \leq 0$, problème de permissions ;
— utiliser `try/except/finally` ou `with` pour garantir la bonne fermeture du fichier.

Exercice 18

(Renommage et déplacement sécurisés de fichiers)
Écrire un script qui demande à l'utilisateur :
— un nom de fichier source ;
— un nouveau nom ;
— éventuellement un nouveau dossier de destination.
Le script tente de renommer (et éventuellement déplacer) le fichier à l'aide de `os.rename`.
Gérer explicitement : `FileNotFoundError`, `PermissionError`, et `OSError` générique.
Afficher un message de succès ou d'échec détaillé, sans laisser le script se terminer brutalement en cas d'erreur.

Exercice 19

(Construction d'un index de fichiers avec gestion d'erreurs)
Écrire un programme qui prend une liste de noms de fichiers texte (saisie par l'utilisateur ou stockée dans un fichier `"liste_fichiers.txt"`) et construit un fichier `"index.txt"` contenant pour chaque fichier : son nom, son nombre de lignes, son nombre de mots.
Pour chaque fichier :
— s'il est lisible, calculer les statistiques et les ajouter à `"index.txt"` ;
— s'il est introuvable ou illisible, écrire dans `"index.txt"` une ligne d'erreur correspondante.
Utiliser des blocs `try/except` pour chaque fichier, afin qu'une erreur sur un fichier n'empêche pas l'indexation des autres.

Exercice 20

Créer une exception personnalisée `QuotaDepasseError`.
Supposons qu'un utilisateur a un quota fictif de Q octets. Écrire un programme qui crée plusieurs fichiers (par exemple `"f1.txt"`, `"f2.txt"`, ...) en écrivant du texte dedans.
Après chaque écriture, estimer la taille totale écrite (en utilisant `len(texte)` ou `os.path.getsize`).
Si la taille cumulée dépasse le quota Q, lever `QuotaDepasseError`, arrêter les écritures et afficher un message indiquant quels fichiers ont été créés avant le dépassement de quota.
Utiliser `try/except/finally` pour s'assurer que tous les fichiers ouverts sont correctement fermés.