Création d'une CLI de couplage d'enregistrements CSV

Une interface en ligne de commande (CLI) est un moyen d'interagir avec un programme en utilisant des lignes de texte plutôt qu'une interface graphique. Elle permet à l'utilisateur de donner des commandes directement à l'application en saisissant des lignes de texte dans le terminal.

Python est très adapté au développement d'outils CLI, c'est à dire paramétrables à l'aide d'un terminal, pour réaliser automatiser des tâches répétitives de manière rapide et efficace.

Vous utilisez déjà de nombreuses CLI au quotidien : la console Python en est par exemple une.

Au programme Cette nouvelle séquence va être l'occasion d'intégrer le mécanisme de couplage développé dans la sequence 1 pour en faire un véritable outil réutilisable.

Un aperçu de l'outil en ligne de commande complet :

```
usage: cli_couplage.py [-h] [-s SEUIL] [-v] fichier1 fichier2
fichier_sortie

Un utilitaire simple de couplage d'enregistrements.

positional arguments:
  fichier1              Premier fichier d'enregistrements à coupler.
  fichier2              Second fichier d'enregistrements à coupler.
  fichier_sortie        Fichier de sortie contenant la jointure couplée des
deux fichiers d'entrée.

options:
  -h, --help            show this help message and exit
  -s SEUIL, --seuil SEUIL
                        Seuil pour le couplage approximatif.
  -v, --verbose         Afficher les détails du couplage.
```

Objectifs

- se familiariser avec la lecture & écriture de fichiers en Python;
- apprendre à manipuler des données structurées de type CSV;
- savoir concevoir et réaliser une CLI à l'aide de la bibliothèque Python `argparse`;
- se familiariser avec les métriques d'évaluation qualité usuelles;

Point d'entrée principal

Une CLI Python simple n'est rien de plus qu'un fichier de script doté (1) d'un point d'entrée principal et (2) d'une manière d'interagir avec l'utilisateur. Lorsque qu'on exécute un script Python, l'interpréteur exécute

les instructions dans l'ordre de lecture, jusqu'à la fin du fichier.

Cela implique donc que les toute variable ou fonction utilisée ait été déclarée avant ! Le code suivant est donc valide...

```python
def hello_world() -> None:
    print("Hello, world!")

hello_world()
# > Hello, world!
```

...mais pas celui-ci :

```python
hello_world()
# Exception !
# > Traceback (most recent call last):
# >   File "<stdin>", line 1, in <module>
# > NameError: name 'hello_world' is not defined

def hello_world() -> None:
    print("Hello, world!")
```

Lorsqu'un script commence devenir complexe et long, cette contrainte nuit fortement à la lisibilité du code et à sa maintenabilité, puisqu'il faut s'assurer que tout est déclaré et utilisé "dans l'ordre"[1]. Les fonctions principales d'un script se retrouve à la fin, ce n'est pas idéal !

Heureusement, il existe une solution simple. L'interpréteur Python fournit une variable système nommée `__name__`, qui contient le texte `"__main__"` **si et seulement si** lorsqu'un script est exécuté comme programme principal, et pas importé comme dépendance dans un autre script. On peut alors encapsuler notre script dans une fonction (par ex. `main()`), et laisser au niveau global une seule instruction vérifiant que le script est exécuté comme programme principal. Si c'est le cas, on appelle la fonction `main`.

```python
# script.py

def main():
    # La logique du script
    fonction_a()
    ...
    fonction_b()

def fonction_a():
    ...

def fonction_b():
    ...
```

```
if __name__ == "__main__":
    main()
```

On voit dans cet exemple que cette technique permet d'organiser le code plus logiquement, avec les fonctions principales en haut du fichier, puis le détail ensuite. On appelle parfois cette organisation *Newspaper code structure* : comme dans un journal, on place les gros titres au début, et le dernier concours du cri de cochon en page 10 !

Cette structuration est très pratique pour mettre en place une CLI, car elle permet notamment de placer en début de script la logique d'interface avec l'utilisateur.

✏️ **Q1. main** ❓ Ouvrez le fichier `sequence_2/cli_couplage.py` et implémentez un point d'entrée nommé `main`, qui imprime "Je suis exécuté comme programme principal !" lorsque pour exécutez le script

```
python couplage.py
```

[1]: Cela pose problème également quand un fichier python doit pouvoir être utilisé à la fois comme script - donc exécuté directement - ou pouvoir être importé comme module (avec la directive `import`).

Lecture des arguments passés par l'utilisateur

Tout l'intérêt d'une CLI repose dans sa capacité à interpréter des paramètres qui lui sont passés lors de son appel par un utilisateur. Il existe plusieurs bibliothèques logicielles en Python pour réaliser cette tâche. Les deux plus populaires sont **`Click`** et **`argparse`**. **`argparse`** fait partie des bibliothèques Python standard qui fournit une approche déclarative pour définir et analyser les arguments en ligne de commande. Il permet de spécifier les arguments attendus, les options, les actions associées, etc. **`Click`** est une bibliothèque externe beaucoup plus expressive et puissante. Elle exploite cependant des aspects avancés de Python pour cacher la complexité de certains mécanismes, la rendant quelque peu "magique". Par conséquent on utilisera dans cet atelier la bibliothèque **`argparse`**, tout à fait suffisante ici.

Comme son nom l'indique, **`argparse`** est une bibliothèque de *parsing* d'arguments, c'est à dire qu'elle fournit des outils pour déclarer, lire et analyser les entrées utilisateur. Son composant principal est la classe `argparse.ArgumentParser`, qui va se charger pour nous:

1. de documenter les paramètres qui peuvent être passés au script;
2. de les analyser lorsque le script est exécuté et de les rendre accessible à l'intérieur du script;
3. de communiquer avec l'utilisateur en cas de problème.

ℹ️ Bloqué(e)s ? Avant tout, cherchez une réponse dans la documentation de **`argparse`** https://docs.python.org/3/library/argparse.html !

✏️ **Q2 : créer un analyseur d'arguments.** Déclarez l'import de argparse, puis déclarez une variable `parser` dans la fonction `main` qui contient une instance de la classe `argparse.ArgumentParser`. Notez que vous pouvez donner au constructeur de `ArgumentParser` un argument `description` qui servira à afficher à l'utilisateur l'objectif de votre programme.

Si vous exécutez le script maintenant, rien ne se passe. En effet, vous venez de créer un analyseur d'arguments, mais il faut lui dire quand le faire ! Pour cela, on doit appeler la méthode `parse_args()` de l'objet `parser`, qui renvoie un objet `Namespace` contenant les arguments analysés.

❓ Ajoutez l'appel à cette méthode à la suite de la création de l'analyseur, puis ré-exécutez le script. Toujours rien ? Et en appelant le script avec l'argument `--help` ?

```
python ./test.py  --help
```

✏️ **Q3 : ajout d'arguments.** N'oublions pas que l'objectif est de réaliser un outil de couplage des enregistrements de deux jeux de données en CSV. Il faut donc donner la possibilité aux utilisateurs de passer au script les chemins vers les deux fichiers. On doit fournir la déclaration de ces arguments à l'analyseur `parser` pour qu'il soit capable de les lire. Déclarer de nouveaux arguments se fait à l'aide de la méthode `parser.add_argument(...)`. Lisez la documentation de cette méthode (https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.add_argument) :

- comprenez-vous la différence entre arguments positionnels et arguments optionnels ?
- ici, quel type d'argument vous semble le plus pertinent ?

❓ Déclarez deux arguments `fichier1` et `fichier2` de type `str` **avant l'appel à `parse_args`**. Ajoutez une note descriptive pour chacun avec le paramètre `help` de `add_argument`.

❓ Qu'obtenez vous maintenant avec `python ./test.py --help` ? Et si vous essayez d'exécuter le script ?

```
python ./test.py
```

✏️ **Q4 : lecture des arguments.** C'est donc l'appel à `parser.parse_args()` qui déclenche effectivement l'analyse des arguments. Cette méthode renvoie un objet de type `Namespace`, qui contient les valeurs des arguments `fichier1` et `fichier2`.

❓ Stockez l'objet retourné par `parse_args` dans une variable nommée `args`. Vous pouvez ensuite accéder aux valeurs des arguments de la manière suivante :

```
def main():
    ...
    args = parser.parse_args()
    fichier1 = args.fichier1
```

❓ Faites en sorte qu'à l'exécution, votre commande `couplage.py` affiche avec `print` les noms des deux fichiers donnés en arguments.

Lecture de fichiers structurés en CSV

Les arguments `fichier1` et `fichier2` pointent en principe vers deux fichiers CSV que l'on cherche à coupler. Mais encore faut-il pouvoir accéder à leur contenu !

Cela peut être fait en deux temps. D'abord, on accède et on ouvre les fichiers en lecture, puis on utilise la bibliothèque Python standard `csv` pour lire leur contenus.

✏️ **Q5 : ouverture des fichiers.** Python fournit nativement la fonction `open()` qui permet de créer un flux de lecture ou d'écriture vers un fichier. La syntaxe est la suivante :

```
# Le second paramètre est le mode d'accès au fichier. "r" signifie "ouvrir
en lecture pour du texte" (read). Un autre mode utile est "w" (write) pour
écrire dans un fichier.
open("chemin/absolu/ou/relatif/vers/un/fichier", "r")
```

Bien qu'on puisse écrire `obj_fichier = open("mon_fichier", "r")`, il s'agit d'une mauvaise pratique que l'on évitera. En effet, comme son nom l'indique, `open` ouvre le fichier, mais ne le referme pas. C'est au développeur de s'assurer de cette fermeture, au risque de fuites mémoire ou bien même de corrompre le fichier. Heureusement, il existe en Python un type d'objet nommé "gestionnaire de contexte" (*context manager*), qui a la particularité de pouvoir être encapsulé dans un bloc de code initié avec l'instruction `with`. Leur syntaxe est la suivante :

```
with open("un_fichier", "r") as objet_fichier:
    # On entre dans un nouveau contexte : "un_fichier" est ouvert en
lecture !
    # Lecture du contenu du fichier, opérations, etc.
    ...
# On sort du contexte, on est assurés que "un_fichier" est bien fermé !
# Essaye de lire de nouveau objet_fichier créera une exception Python.
```

Ici, le gestionnaire de contexte garantit que le fichier sera bien fermé lorsque l'interpréteur aura terminé d'exécuter les instructions comprises dans le bloc de code `with`.

❓ Créez une fonction `charger_enregistrements` qui prend en paramètre le chemin vers un fichier, et l'ouvre en lecture à l'aide d'un bloc `with`. Pour le corps du bloc, passez à la question suivante.

✏️ **Q6 : lecture de fichiers CSV.** Le format CSV (*comma separated values*) permet de décrire des tableaux de données sous forme de texte. Chaque ligne du fichier est une ligne du tableau, et les cellules sont séparées par un caractère spécial, traditionnellement une virgule.

Python est fourni avec la bibliothèque standard `csv` dédiée à la lecture et à l'écriture de ce type de fichier. Ajoutez son import à `couplage.py`. Elle fournit notamment la fonction `reader(objet_fichier, delimiter=...)` qui prend en paramètre un objet fichier tel que renvoyé par `open()`, et un paramètre optionnel `delimiter` pour fixer le séparateur de cellules. Cette fonction permet de lire un fichier CSV ligne par ligne sous la forme d'une liste de champs.

❓ Ajoutez au bloc `with` de la fonction `charger_enregistrements` les instructions pour lire le contenu du CSV ouvert et renvoyer une liste contenant l'ensemble des lignes du fichier.

⚠️ Attention ! la première ligne contiendra les noms des colonnes !

ℹ️ `reader()` ne renvoie pas une liste mais un itérateur; vous aurez besoin de le convertir en liste explicitement : `list(reader(...))`

❓ Appelez `charger_enregistrements` dans la fonction `main` pour récupérer le contenu des deux fichiers en arguments et imprimez avec `print` le nombre de lignes de chacun, ainsi que leurs 2 premières lignes.

⚠️ Attention ! séparez les noms des colonnes du contenu des CSV en deux variables distinctes, par exemple :

```
enregistrements_1 = charger_enregistrements(...)
enregistrements_1_header = enregistrements_1[0] # Première ligne
enregistrements_1 = enregistrements_1[1:] # Le reste

# Idem pour enregistrements_2 ...
```

❓ Testez votre commande pour lire les fichiers `sequence_2/data/didot_1842_small.csv` et `sequence_2/data/didot_1843_small.csv`.

Couplage de fichiers CSV

Le squelette de l'outil de couplage en ligne de commande est en place; reste maintenant à y insérer la logique de couplage. Heureusement, une partie des mécanismes a déjà été implémenté dans la séquence précédente !

Certaines adaptations sont tout de même nécessaires. Par exemple, la fonction `couplage` précédemment implémentée prenait en paramètre seulement 2 enregistrements, or il y a maintenant 2 ensembles d'enregistrements à coupler.

✏️ **Q7 : reprise de code.** ❓ Copiez dans `couplage.py` 1 les fonctions suivantes créées dans la séquence n°1 :

- `score_exact(list[str], list[str]) -> bool:`
- `score_approximatif(list[str], list[str]) -> float`
- `decision(float, float) -> bool`
- `normalisation(str) -> str` N'oubliez pas d'ajouter les dépendances nécessaires (à nltk, etc.) Toutes les briques de base sont maintenant prêtes à être assemblées.

✏️ **Q8 : processus principal.** Le processus de couplage entre deux ensembles d'enregistrements peut être décomposé ainsi :

1. **[Pré-traitements]** : normalisation des champs utiles au couplage ;
2. **[Comparaison]** : calcul du score de couplage entre toutes les paires possibles d'enregistrements ;

3. **[Classification]** : prise de décision pour chaque paire : est un couplage valide (*match*) ou non (*non-match*) ;
4. **[Évaluation]** : calculs d'indicateurs de qualité du couplage réalisé ;

Voici le pseudo-code d'une fonction `couplage_multi` qui réalise les étapes 1 à 3. Laissons l'étape 4 de coté pour le moment. ❓ Implémentez cette fonction dans `couplage.py`, en vous appuyant sur la fonction `couplage` implémentée dans la séquence n°1.

```
_____________________________________________________________________
Algorithme 1 : couplage de deux ensembles d'enregistrements
Résultat : une liste de couples d'enregistrements.
Entrées :
    - E1 et E2 : deux listes d'enregistrements.
    - normalisation : un booléen indiquant s'il faut réaliser une étape de
normalisation des champs des enregistrements.
    - exact_match un boolean indiquant s'il faut réaliser un couplage exact
ou approximatif
Sorties : une liste de couples (i, j, s) où i et j sont les indices des
enregistrements dans leur listes respectives, et s le score de couplage.
_____________________________________________________________________

Initialiser une liste vide 'couples'.

SI exact_match est vrai ALORS
    forcer le seuil à 0

SI normalisation est vrai ALORS
    POUR chaque enregistement e dans E1 (resp. E2) FAIRE
        normaliser les champs de e
    FIN POUR
FIN SI

POUR chaque enregistrement e1 dans E1 et chaque enregistrement e2 dans E2
FAIRE
    calculer score de couplage e1 et e2, exact ou approximatif
    SI il s'agit d'un *match* ALORS
        créer une paire (i, j, s), i étant l'indice de e1 dans E1, j celui
de e2 dans E2, s le score, et ajouter cette paire à la liste de couples
    SINON
        continuer
    FIN SI
FIN POUR

RETOURNER couples
```

❓Ajoutez ensuite l'appel à cette fonction dans le corps de la fonction `main` pour qu'elle s'exécute sur les ensembles d'enregistrements obtenus avec `charger_enregistrements`.

❓Créez maintenant une fonction nommée `affiche_couplage` qui prend en paramètre les 2 ensembles d'enregistrements ainsi que le résulta d'un couplage, et affiche :

- le nombre de couples trouvés ;
- la proportion d'enregistrements du premier et du second ensembles qui ont été couplés (attention aux enregistrements couplés plusieurs fois !) ;
- la liste de tous les couples avec le formatage suivant :

```
{index_1}: {enregistrement_1} --[MATCH {score}]-- {index_2}:
{enregistrement_2}

# Exemple :
# 0: ['Abadie', 'coiffeur', '21 Miroménil'] --[MATCH 1.0]-- 1: ['Abadie',
'coiffeur', '21 Miroménil']
```

Faites en sortes qu'elle soit appelée par la fonction `main` pour imprimer les résultats du couplage.

❓ Testez ensuite sur `sequence_2/data/didot_1842_small.csv` et `sequence_2/data/didot_1843_small.csv` en utilisant un couplage exact. Combien de liens trouvez-vous ? Vérifiez en ouvrant les deux fichiers CSV que ces liens paraissent corrects.

✏️ **Q9 : argument optionnel : couplage approximatif** Un argument optionnel est tout simplement un argument qui ne déclenchera pas une erreur de l'analyseur s'il n'est pas spécifié.

Ce type d'argument est extrêmement utile pour construire des commandes modulaires, ainsi que pour réécrire des paramétrages par défaut.

Le seuil de couplage approximatif est un bon candidat. En effet, en faire un argument optionnel permet le comportement suivant :

- s'il est absent, la commande réalise un couplage exact;
- s'il est donné, la commande réalise un couplage approximatif.

Pour **`argparse`**, il s'agit d'un argument optionnel, déclaré avec `add_argument` (https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.add_argument).

❓ Ajoutez un argument optionnel, dont le nom court est `-s` et le nom long `--seuil`, de type `float`, qui permet de spécifier un seuil de décision pour le couplage approximatif. Adaptez ensuite le code des fonctions `main` et `couplage_multi` pour basculer entre couplage exact et couplage approximatif selon si l'argument de seuil est présent ou non. Vérifiez que cela fonctionne bien !

ℹ️ La valeur d'un argument optionnel absent sera None.

✏️ **Q10 : *flag arguments***. Les arguments *flags* sont un type particulier d'arguments optionnels. Il s'agit de d'arguments booléens, qui ne prennent pas de valeur et servent en quelques sortes de boutons *On/Off*. Ce type d'argument est utile pour qu'un utilisateur puisse agir sur l'activation ou la désactivation de mécanismes d'une commande. Un *flag* courant est `-h` (`--help`), comme dans `python ./couplage.py --help`. S'il est présent, **`argparse`** affiche l'aide auto-générée de la commande.

On trouve aussi souvent le *flag* `-v`, ou `--verbose`, pour signaler à la commande qu'on souhaite qu'elle soit plus "bavarde", c'est à dire qu'elle affiche d'avantage d'informations.

Un *flag argument* s'implémente de la même manière qu'un argument optionnel classique, mais il faut en plus passer à la méthode `add_argument` le paramètre `action='store_true'`. Cela indique l'action que l'analyseur doit appliquer s'il rencontre cet argument; ici stocker la valeur booléenne `true`.

❓ **bonus (optionnel)** Faites en sortes que `affiche_couplage` ne soit appelée que si un *flag* `-v / --verbose` est spécifié.

❓ **bonus (optionnel)** Ajoutez le *flag* argument `--no-normalisation` qui, s'il est spécifié, désactive l'étape de normalisation des chaînes de caractères.

✏️ **Q11 : *export du couplage.***

Reste à exporter le résultat du couplage, dans un format utilisable par un être humain.

Une possibilité consiste à écrire un nouveau fichier CSV contenant la jointure des deux CSV couplés, ainsi que le score de liage.

Prenons l'exemple suivant :

- `fichier_1.csv`:

	per	act	loc
0	Regnier	papetier	boul. des italiens
1	Soret	poterie de terre	b. des Four-neaux 21

- `fichier_2.csv`:

	per	act	loc
0	Simon	tireur d'or	Geoffroy-l'Angevin 7
1	Regnier	papetier	boul. des italiens

- couplage : `[(0, 1, 0.0)]`

Une solution simple consiste à n'exporter que les enregistrement liés, pour former la table suivante

	id_1	per_1	act_1	loc_1	id_2	per_1	act_1	loc_1	score
0	0	Regnier	papetier	boul. des italiens	1	Regnier	papetier	boul. des italiens	0.0

❓ Ajoutez un nouvel argument positionnel nommé `fichier_sortie` à votre commande, de type `str`, qui sera le chemin du CSV exporté.

❓ Créez une fonction `jointure_couplage`, qui prend en entrée deux listes d'enregistrements ainsi qu'un résultat de couplage, et retourne la liste jointe des enregistrements couplés, plus le score de couplage. Vous pouvez vous appuyer sur l'exemple suivant qui réalise la jointure de deux enregistrements :

```python
def exemple(enregistrement1: list[str], enregistrement2: list[str],
couplage: tuple[int, int, float] ) -> list:
    jointure = []
    # La méthode list.extend() permet d'ajouter tous les éléments d'une
liste à une autre liste
    # On ajoute d'abord les éléments de l'enregistrement 1, puis ceux de
l'enregistrement 2, puis le score de couplage
    jointure.extend(enregistrement1)
    jointure.extend(enregistrement2)
    jointure.append(couplage[2])
    return jointure

# Test
a = ["a", "b", "c"]
b = ["a", "b", "c"]
c = (0, 1, 0.0)
print(exemple(a, b, c))
# > ['a', 'b', 'c', 'a', 'b', 'c', 0.0]
```

❓ Créez une fonction `export_csv` qui prend en paramètres:

- `jointure` : le résultat de `jointure_couplage()`
- `colonnes` : une liste de noms de colonnes
- `fichier_sortie` : le chemin du fichier CSV à créer.

Cette fonction doit contenir un bloc `with` similaire à la fonction `charger_enregistrements`, mais cette fois le fichier est à ouvrir en mode écriture ("w").

L'écriture du contenu du CSV peut être ensuite réalisé de la manière suivante:

```python
with open(...) as fichier:
    writer = csv.writer(fichier)
    writer.writerow(colonnes)
    writer.writerows(jointure) # Attention à la distinction entre writerow
et writerows !
```

❓ Testez votre commande pour vérifier que l'export fonctionne correctement !

Bonus : évaluation du couplage

...