Module Scraper

Introduction

Le module `Scraper` est conçu pour scraper les données des sites web en suivant une série d'états et de méthodes définies dans un fichier de configuration. Il gère diverses étapes du processus de scraping, y compris l'interaction avec la page web, la gestion des pilotes, l'extraction des données, et l'interaction avec une API si nécessaire. Le processus est contrôlé et suivi à l'aide de la gestion des logs et des états. Le module s'appuie sur d'autres classes auxiliaires comme :

- `DataHandler`
- `ScrapingEngine`
- `ApiHandler`
- `DriverHandler`
- `PageInteractor`
- `UnifiedDataExtractor`
- `Controller`

Spécificités et Capacités

Numéro	Spécificité / Capacité	Classe / Module
1	Initialisation du Scraper	`Scraper`
2	Gestion des Étapes de Scraping	`ScrapingEngine`
3	Interaction avec les Pages Web	`PageInteractor`
4	Gestion du Pilote	`DriverHandler`
5	Extraction et Formatage de Données	`UnifiedDataExtractor`
6	Manipulation des Données	`DataHandler`
7	Interaction avec l'API	`ApiHandler`
8	Contrôle du Scraper	`Controller`
9	Gestion des Logs	`logging`
10	Mise à Jour du Fichier de Configuration	`Scraper`
11	Information sur l'État du Scraper	`Scraper`

Configuration du Fichier

La configuration du fichier joue un rôle crucial dans le fonctionnement du module `Scraper`. Voici les composants principaux d'un fichier de configuration:

Composant	Description

Composant	Description
`data_sources`	Une section contenant des informations sur les sources de données. Ces données peuvent être obtenues (get) et définies (set), et une recherche peut être effectuée dessus. Il y a aussi une clé `saved_data` où les données sont sauvegardées.
`base_url`	L'URL de base pour la navigation ou l'extraction de données.
`initial_state`	L'état initial dans une séquence d'états pour l'automatisation.
`userAgent`	L'agent utilisateur à utiliser lors de la navigation.
`browser`	Le navigateur à utiliser (par exemple, Chrome).
`default_cases`	Des cas par défaut pour gérer des méthodes de steps. Si une méthode a en général des paramètres répétitifs, on déclare le cas par défaut, et ce dernier est fusionné avec la step (la step écrase si les mêmes paramètres sont présents).
`states`	Une section décrivant différents états et transitions pour l'automatisation. `initial_state` étant l'état initial, chaque step a un nom, une méthode et ses paramètres, un `next_state`, et si le résultat de la step est évaluable, on peut définir deux steps: true et false.

Exemple de Configuration

```
# Configuration de base pour la navigation web
base_url: https://www.example.com # L'URL de base pour la navigation web
userAgent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/88.0.4324.150 Safari/537.36 # Chaîne de l'agent utilisateur
browser: chrome # Choix du navigateur, par exemple, Chrome

# Section des sources de données pour stocker et manipuler les données pendant le
scraping
data_sources:
  records: [] # Une liste pour stocker les enregistrements
  custom_dict: {} # Un dictionnaire pour stocker les paires clé-valeur
  custom_list: [] # Une liste pour stocker des données personnalisées
  saved_data: {} # Un dictionnaire spécial pour stocker les données sauvegardées

# L'état initial où commence l'automatisation
initial_state: init_driver

# Cas par défaut communs à plusieurs étapes
default_cases:
  common_handler: # Un gestionnaire commun par défaut
    common_param: "value"
    data: ''
    common_param_2:
          Content-Type: "application/json"


# Les états définissent la séquence d'actions et de transitions
states:
```

```yaml
  init_driver: # Configuration initiale du pilote
    method: initialize
    next_state: do_something

  do_something: # Une étape exemple avec des états suivants conditionnels
    method: action
    parameters:
      key: value
    next_state:
      true: success_step # Si le résultat est vrai, aller à success_step
      false: failure_step # Si le résultat est faux, aller à failure_step

  success_step: # Une étape pour gérer le succès
    method: data_handler_step
    next_state: finish
    parameters:
      action: set_data
      path: data_sources.saved_data.success_result # Chemin vers saved_data
      value: $$step_results.do_something.success_result # Utilisation de $$ pour
obtenir les données

  failure_step: # Une étape pour gérer l'échec
    method: data_handler_step
    next_state: finish
    parameters:
      action: set_data
      path: data_sources.saved_data.failure_result # Chemin vers saved_data
      value: $$step_results.do_something.failure_result # Utilisation de $$ pour
obtenir les données

  save_config: # Mise à jour des données sauvegardées dans le fichier de
configuration
    method: update_config_file
    next_state: finish # État final de la séquence

  use_default_case: # Étape utilisant les valeurs de default_cases telles qu'elles
sont
    method: common_handler # Utilise le cas par défaut 'common_handler'
    next_state: use_default_case

  override_default_case: # Étape écrasant certains paramètres de default_cases
    method: common_handler # Utilise le cas par défaut 'common_handler'
    next_state: finish
    parameters:
      common_param: "new_value" # Écrase le paramètre commun
      data: "%%previous_result%%" # Utilise %%previous_result%% pour accéder
directement au résultat de l'étape précédente
```

Composants spécifiques

- `states`

La section des états décrit les différentes étapes et transitions pour l'automatisation. Chaque étape a un nom, une méthode, des paramètres et un `next_state`.

```
# Exemple illustrant la structure d'une étape dans `states`
init_driver:
  method: initialize
  next_state: do_something
```

Le `next_state` peut être conditionnel, permettant de définir deux steps (true et false) en fonction de l'évaluation du résultat de l'étape.

```
# Exemple illustrant une étape conditionnelle dans `states`
do_something:
  method: action
  parameters:
    key: value
  next_state:
    true: success_step # Si le résultat est vrai, aller à success_step
    false: failure_step # Si le résultat est faux, aller à failure_step
```

- `default_cases`

```
use_default_case:
  method: common_handler # Utilise le cas par défaut 'common_handler'
  parameters:  # Ajoutez les paramètres nécessaires ici
    common_param: valeur1
    data: ''
    common_param_2: -1
```

Les cas par défaut sont définis pour gérer des méthodes de steps qui ont généralement des paramètres répétitifs. Au lieu de répéter ces paramètres dans chaque étape, on déclare un cas par défaut, et ce dernier est fusionné avec la step.

```
# Exemple illustrant l'utilisation de `default_cases`
override_default_case:
  method: common_handler # Utilise le cas par défaut 'common_handler'
  next_state: finish
  parameters:
    common_param: "nouvelle_valeur" # Écrase le paramètre commun
    data: "%%previous_result%%" # Utilise `%%previous_result%%` pour accéder
directement au résultat de l'étape précédente
```

Si les mêmes paramètres sont présents dans la step, ils écrasent les valeurs par défaut. Par exemple, le `common_handler` dans `default_cases` contient des paramètres qui peuvent être utilisés dans plusieurs étapes (`use_default_case` et `override_default_case`), et ils peuvent être écrasés si nécessaire.

```
# Exemple illustrant l'utilisation de `default_cases` avec écrasement de
paramètres
override_default_case:
  method: common_handler # Utilise le cas par défaut 'common_handler'
  next_state: finish
  parameters:
    common_param: "nouvelle_valeur" # Écrase le paramètre commun
    data: "%%previous_result%%" # Utilise `%%previous_result%%` pour accéder
directement au résultat de l'étape précédente
```

Sources de Données

La section des sources de données joue un rôle crucial dans le processus de scraping du module `Scraper`. Elle permet de stocker, manipuler et accéder aux données essentielles tout au long du processus de scraping et d'automatisation. Cette section est un élément central qui facilite la gestion des données collectées et permet leur utilisation à différentes étapes du processus.

- `data_sources`

La section `data_sources` offre la possibilité de déclarer des variables personnalisées qui peuvent être utilisées tout au long du processus de scraping et d'automatisation. Ces variables peuvent contenir des informations pertinentes pour le processus, telles que des valeurs constantes, des sélecteurs d'éléments HTML, des URLs spécifiques, etc. En définissant ces variables, vous pouvez les réutiliser facilement à travers différentes étapes et méthodes, simplifiant ainsi la personnalisation et la gestion des opérations.

```
# Exemple de déclaration de variables dans `data_sources`
data_sources:
  records: []
  custom_dict:
    key: "valeur_de_test"
  custom_list: [1, 2, 3]
  saved_data: {}
```

- `saved_data`

La clé `saved_data` dans la section des sources de données `data_sources` joue un rôle particulier dans le processus d'automatisation. Cette clé est spécialement conçue pour stocker des données sauvegardées lorsqu'une étape est exécutée avec la méthode `save_config`. Ces données sauvegardées sont ensuite écrites dans le fichier de configuration. Les données sauvegardées peuvent être utilisées ultérieurement dans un autre processus de scraping ou d'automatisation. Cette fonctionnalité permet de capturer et de conserver les résultats intermédiaires, les informations importantes ou les états spécifiques lors de différentes étapes.

```yaml
# Exemple illustrant l'utilisation de la clé `saved_data` dans `data_sources`
data_sources:
  saved_data:
    int_value: 5 # Exemple de données sauvegardées de type entier

states:
  change_int_value:
    method: data_handler_step
    next_state: save_config
    parameters:
      action: set_data
      path: data_sources.saved_data.int_value # Chemin vers l'élément sauvegardé
      value: 10 # Nouvelle valeur pour l'élément sauvegardé

  save_config:
    method: save_config_step
    next_state: finish #Après cette étape, le fichier de configuration sera
sauvegardé avec le nouveau int_value.
```

Dans cet exemple, nous avons une clé `saved_data` dans la section `data_sources` où nous stockons `int_value`. Ensuite, nous avons une étape `change_int_value` qui utilise la méthode `set_data` de `data_handler_step` pour modifier la valeur de cet entier. La nouvelle valeur est définie comme `10`. Après cette étape, une autre étape `save_config` exécute la méthode `save_config_step`, ce qui entraîne la sauvegarde du fichier de configuration avec la nouvelle valeur de `int_value`.

```yaml
# Exemple apres run
data_sources:
  saved_data:
    int_value: 10 # Exemple de données sauvegardées de type entier
states:
 ...
```

L'utilisation de `saved_data` peut contribuer à rendre vos opérations de scraping et d'automatisation plus flexibles et mieux organisées, en vous permettant de partager des données entre différentes parties et etapes du processus.

- `step_results`

La section `step_results` joue un rôle similaire à celui de `data_sources`, mais elle est spécifiquement conçue pour stocker les résultats retournés par les différentes étapes du processus d'automatisation. Cela permet de capturer et de conserver les données générées ou extraites à chaque étape, ce qui peut être essentiel pour le suivi, le débogage et la réutilisation de ces données.

```yaml
states:
```

```
  fetch_data: # Étape pour obtenir des données
    method: fetch_data_from_api
    next_state: process_data

  process_data: # Étape pour traiter les données obtenues
    method: process_data_function
    parameters:
      data: "$$step_results.fetch_data.response_data" # Utilisation des données de
l'étape précédente
    next_state: update_database

  update_database: # Étape pour mettre à jour la base de données
    method: update_database_function
    parameters:
      processed_data: "$$step_results.fetch_data.response_data" # Utilisation des
données de la première étape
    next_state: finish
```

Utilisation des données dynamiques

Dans les paramètres, il est possible d'utiliser des données dynamiques en utilisant la syntaxe `$$`. Par exemple, dans les étapes `success_step` et `failure_step`, la valeur est obtenue à partir du résultat d'une autre étape en utilisant `$$step_results.do_something.success_result`.

```
# Exemple illustrant l'utilisation de données dynamiques
 success_step:
  method: data_handler_step
  next_state: finish
  parameters:
    action: set_data
    path: data_sources.saved_data.success_result
    value: $$step_results.do_something.success_result #La valeur stockée dans
data_sources.saved_data.success_result
```

De plus, il est possible d'accéder rapidement au résultat de l'étape précédente en utilisant la syntaxe `%%previous_result%%`.

```
# Exemple illustrant l'utilisation de %%previous_result%%
override_default_case:
  method: common_handler
  next_state: finish
  parameters:
    common_param: "nouvelle_valeur"
    data: "%%previous_result%%" #Utilise %%previous_result%% pour accéder
directement au résultat de l'étape précédente
```

Étapes avec la Classe Scraper

La classe Scraper que vous avez mise en place offre des méthodes pour exécuter des actions qui sont spécifiques à la gestion et au contrôle du processus d'automatisation du scraping:

Méthode	Description
`get_scraper_info`	Récupère des informations sur l'état actuel du scraper, le nombre d'enregistrements collectés, le fichier de configuration utilisé, etc.
`pause`	Met en pause le processus d'automatisation à un moment donné, permettant des vérifications ou des interactions manuelles avant de poursuivre.
`update_config_file`	Met à jour le fichier de configuration avec les sources de données actuelles, reflétant ainsi les modifications apportées pendant le processus d'automatisation.

Exemples

`get_scraper_info`

Description: Récupère des informations sur l'état actuel du scraper.

Configuration:

```
states:
  check_scraper_status:
    method: get_scraper_info
    next_state: finish
```

`pause`

Description: Met en pause le processus d'automatisation, permettant des vérifications ou des interactions manuelles.

Configuration:

```
states:
  after_scrape:
    method: pause
    next_state: process_data
```

`update_config_file`

Description: Met à jour le fichier de configuration avec les sources de données actuelles.

Configuration:

```
states:
  update_configuration:
    method: update_config_file
    next_state: finish
```

En utilisant ces méthodes, vous pouvez contrôler le flux de votre processus d'automatisation, obtenir des informations importantes sur son fonctionnement, faire des pauses pour des vérifications ou des interactions manuelles, et mettre à jour votre configuration en fonction des données collectées. Ces actions vous donnent un contrôle plus granulaire sur le processus d'automatisation, tout en utilisant les fonctionnalités fournies par la classe Scraper.

Étapes avec la Classe DriverHandler

La classe `DriverHandler` gère les interactions avec le navigateur web à l'aide de Selenium. Elle fournit une série de méthodes pour initialiser le driver, naviguer vers des liens, passer à la page suivante, nettoyer les cookies, et scraper une page.

Méthodes principales

Méthode	Description	Paramètres	Retour
`init_driver`	Initialise le driver web pour Selenium. Supporte Chrome et Firefox.		Le driver initialisé. ValueError ou WebDriverException si un problème survient.
`goto_link`	Navigue vers le lien spécifié.	`link` - Le lien vers lequel naviguer.	Un dictionnaire contenant l'URL, le HTML, et le titre.
`goto_next_page`	Passe à la page suivante en utilisant l'URL de base et le numéro de page suivant.	`pagenum` (optionnel) - Le numéro de la page actuelle.	Un dictionnaire contenant des informations sur la page.
`scrap_page`	Scraper la page en utilisant la méthode et la valeur données, retourne l'attribut spécifié de l'élément trouvé.	`by_method` - La méthode pour localiser l'élément, `value` - La valeur à utiliser, `return_attr` - L'attribut à retourner.	La valeur de l'attribut spécifié de l'élément trouvé, ou None si erreur.
`cleanup`	Nettoie les cookies et le cache.		

Méthode	Description	Paramètres	Retour
`check_condition`	Vérifie une condition à l'aide d'une fonction.	`condition_func` - La fonction de condition à vérifier, **kwargs - Arguments supplémentaires.	Le résultat de la condition vérifiée.

Exemples

`init_driver`

Description: Initialise le driver web pour Selenium en fonction du navigateur spécifié dans la configuration. Supporte actuellement Chrome et Firefox.

Configuration:

```
init_driver:
  method: driver_handler_step
  next_state: goto_next_page
  parameters:
    action: init_driver
```

Retour:

- `webdriver.Chrome or webdriver.Firefox or None:` Le driver initialisé pour Chrome ou Firefox, ou None si l'initialisation échoue.

Exceptions:

- `ValueError:` Si un navigateur non pris en charge est spécifié.
- `WebDriverException:` Si le driver n'est pas trouvé ou ne peut pas être installé.

`goto_link`

Description: Navigue vers le lien spécifié et retourne diverses informations sur la page.

Configuration:

```
get_and_goto_link_data:
  method: driver_handler_step
  parameters:
    action: goto_link
    link: "$$step_results.get_properties[-1].url"
  next_state: eval_isLinkDead
```

Paramètres:

- `link (str):` Le lien vers lequel naviguer.

Retour:

- `dict or None:` Un dictionnaire contenant des informations sur la page, telles que:
  - `"url":` URL actuelle.
  - `"html":` Code source de la page.
  - `"title":` Titre de la page. Ou None si une erreur s'est produite pendant la navigation.

goto_next_page

Description: Navigue vers la page suivante en utilisant l'URL de base et le numéro de la page suivante.

Configuration:

```
goto_next_page:
  method: driver_handler_step
  parameters:
    action: goto_next_page
  next_state: some_next_state
```

Paramètres:

- `pagenum (int, optional):` Numéro de la page vers laquelle naviguer. Si non spécifié, utilise la valeur actuelle de `pagenum`.

Retour:

- `dict or None:` Un dictionnaire contenant des informations sur la page, telles que:
  - `"url":` URL actuelle.
  - `"html":` Code source de la page.
  - `"title":` Titre de la page.
  - `"current_page":` Numéro de la page actuelle. Ou None si une erreur s'est produite pendant la navigation.

scrap_page

Description: Scraper la page en utilisant la méthode et la valeur données, retourne l'attribut spécifié de l'élément trouvé.

Configuration:

```
scrap_page:
  method: driver_handler_step
  next_state: unified_extract_and_format
  parameters:
    action: scrap_page
    by_method: XPATH
    value: //*[@id="__next"]/div/main/div/div[6]/div[1]/div/div[2]
```

Paramètres:

- `by_method (str):` Méthode pour localiser l'élément. Valeurs : "XPATH", "ID", "NAME", "TAG_NAME", "CLASS_NAME", "CSS_SELECTOR".
- `value (str):` Valeur à utiliser avec `by_method`.
- `return_attr (str, optional):` Attribut à retourner. Défaut : 'outerHTML'. Peut être '*' pour toutes les infos.

Retour:

- `str or None:` Valeur de l'attribut spécifié, ou None si erreur.
- Si `return_attr` est '*', un dictionnaire est retourné avec : "html", "text", "value", "tag", "class", "images", "links".

Exceptions:

- `TimeoutException:` Temps d'attente dépassé.
- `NoSuchElementException:` Élément non trouvé.
- `AttributeError:` Attribut `by_method` inexistant.

`cleanup`

Description: Nettoie les cookies et le cache.

Configuration

```
  clear_cache:
    method: driver_handler_step
    parameters:
      action: cleanup
    next_state: get_properties
```

`check_condition`

Description: Vérifie une condition en utilisant une fonction spécifiée.

Configuration:

```
check_condition:
  method: driver_handler_step
  parameters:
    action: check_condition
    condition_func: "strInDriver"
    strToCheck: "example"
    checkIn: "url"
  next_state: some_next_state
```

Paramètres:

- `condition_func (str):` Nom de la fonction de condition. Valeur actuelle prise en charge : `"strInDriver"`.
- `strToCheck (str, optional):` Chaîne à vérifier dans l'URL, le code HTML ou le titre de la page.
- `checkIn (str, optional):` L'emplacement où vérifier la chaîne. Valeurs possibles : `"url"`, `"html"`, `"title"`.

Retour:

- `bool: True` si la condition est remplie, `False` sinon.

Exceptions:

- `ValueError:` Si la fonction de condition ou l'emplacement de vérification est inconnu.

Utilisation de la méthode `driver_handler_step`

La méthode `driver_handler_step` dans la classe principale joue un rôle clé dans le contrôle du navigateur web, permettant d'exécuter différentes actions de gestion de driver. Elle peut être utilisée pour initialiser le driver, naviguer vers des liens, passer à la page suivante, nettoyer les cookies, et scraper une page.

Comment ça fonctionne

1. **Lecture des paramètres:** La méthode prend en entrée une action spécifiée et des arguments supplémentaires (`**kwargs`). L'action peut être l'une des suivantes: 'init_driver', 'goto_next_page', 'goto_link', ou 'scrap_page'.
2. **Exécution de l'action:** La méthode vérifie si l'action spécifiée existe dans le gestionnaire de driver, puis récupère et exécute cette méthode avec les arguments fournis.
3. **Gestion des exceptions:** Si l'action spécifiée n'est pas reconnue, une exception `ValueError` est levée.
4. **Retour du résultat:** Le résultat de l'action est retourné. Pour l'action 'init_driver', le driver est également enregistré dans l'instance de la classe.

```
def driver_handler_step(self, action, **kwargs):
    """
    Execute a driver handler action.

    Args:
        action (str): The action to execute. Can be 'init_driver',
'goto_next_page', 'goto_link', or 'scrap_page'.
        **kwargs: Additional arguments for the action method.
    """
    if hasattr(self.driver_handler, action):
        method = getattr(self.driver_handler, action)
        result = method(**kwargs)
        if action == 'init_driver':
            self.driver = result
        return result
```

```
        else:
            raise ValueError(f"Unknown action {action}")
```

Cette méthode offre une flexibilité et une simplicité dans la gestion du navigateur web, en encapsulant les différentes actions dans une méthode unique et facilement configurable.

Étapes avec la Classe DataHandler

La classe `DataHandler` gère les interactions avec les données, permettant de récupérer, définir, supprimer et créer des données à partir de chemins spécifiés. Elle utilise l'expression JSONPath pour naviguer dans les structures de données et offre des méthodes pour évaluer des expressions.

Méthodes principales

Méthode	Description	Paramètres	Retour
`get_data`	Récupère les données du chemin spécifié.	`path` (str) - Le chemin des données à récupérer.	Les données récupérées ou `None`.
`set_data`	Définit une valeur au chemin spécifié.	`path` (str), `value` (any) - La valeur à définir.	None.
`delete_data`	Supprime la valeur au chemin spécifié.	`path` (str) - Le chemin des données à supprimer.	None.
`search`	Recherche une clé et retourne toutes les correspondances.	`key` (str) - La clé à rechercher.	Liste des correspondances ou `None`.
`create`	Crée une nouvelle valeur au chemin spécifié.	`path` (str), `value` (any) - La valeur à créer.	La valeur créée.
`evaluate`	Évalue une expression et retourne le résultat.	`expression` (str) - L'expression à évaluer.	Résultat de l'évaluation ou `ValueError`.

Exemples

`get_data`

Description: Récupère les données du chemin spécifié dans la structure de données.

Paramètres:

- `path (str):` Le chemin des données à récupérer.

Retour:

- Les données récupérées, ou `None` si le chemin n'est pas trouvé.

```
get_user_name:
  method: data_handler_step
  next_state: some_next_state
  parameters:
    action: get_data
    path: data_sources.user.name
```

Il est également possible de récupérer les données de manière plus automatisée sans avoir besoin de définir une étape spécifique, en utilisant la syntaxe `$$`. Cette syntaxe permet d'utiliser des données dynamiques à partir du résultat d'une autre étape.

Utilisation des données dynamiques

Dans les paramètres, il est possible d'utiliser des données dynamiques en utilisant la syntaxe `$$`. Par exemple, dans les étapes `success_step` et `failure_step`, la valeur est obtenue à partir du résultat d'une autre étape en utilisant `$$step_results.do_something.success_result`.

```
# Exemple illustrant l'utilisation de données dynamiques
success_step:
  method: data_handler_step
  next_state: finish
  parameters:
    action: set_data
    path: data_sources.saved_data.success_result
    value: $$step_results.do_something.success_result # La valeur stockée dans
data_sources.saved_data.success_result
```

set_data

Description: Définit une valeur au chemin spécifié dans la structure de données.

Paramètres:

- `path (str):` Le chemin où définir la valeur.
- `value (any):` La valeur à définir.

```
set_oldest_page:
  method: data_handler_step
  next_state: save_config
  parameters:
    action: set_data
    path: data_sources.saved_data.oldest_page
    value: $$step_results.goto_next_page.current_page
```

delete_data

Description: Supprime la valeur au chemin spécifié dans la structure de données.

Paramètres:

- `path (str):` Le chemin des données à supprimer.

```
remove_user_info:
  method: data_handler_step
  next_state: process_next
  parameters:
    action: delete_data
    path: step_results.user_info.name
```

search

Description: Recherche une clé dans la structure de données et retourne toutes les correspondances.

Paramètres:

- `key (str):` La clé à rechercher.

Retour:

- Liste des correspondances ou `None` si la clé n'est pas trouvée.

```
search_for_key:
  method: data_handler_step
  next_state: handle_results
  parameters:
    action: search
    key: name
```

create

Description: Crée une nouvelle valeur au chemin spécifié dans la structure de données.

Paramètres:

- `path (str):` Le chemin où créer la nouvelle valeur.
- `value (any):` La valeur à créer.

Retour:

- La valeur créée.

```
create_new_key:
  method: data_handler_step
  next_state: update_data
  parameters:
```

```
    action: create
    path: data_sources.new_key
    value: {"field": "value"}
```

`evaluate`

Description: Évalue une expression et retourne le résultat. Les références aux données peuvent être incluses dans l'expression.

Paramètres:

- `expression (str):` L'expression à évaluer.

Retour:

- Résultat de l'évaluation, ou `ValueError` si l'expression est invalide.

```
evaluate_expression:
  method: data_handler_step
  next_state: apply_result
  parameters:
    action: evaluate
    expression: $$data_sources.count * 10
```

Utilisation de la méthode `data_handler_step`

La méthode `data_handler_step` dans la classe principale joue un rôle essentiel dans la manipulation et le contrôle des structures de données complexes. Elle permet d'exécuter différentes actions pour gérer les données, telles que récupérer, définir, supprimer, créer des données, et évaluer des expressions.

Comment ça fonctionne

1. **Initialisation de DataHandler:** La méthode initialise une instance de la classe `DataHandler` avec les sources de données et les résultats des étapes.
2. **Définition de la carte d'action:** Une carte d'action est définie pour mapper les actions spécifiées avec les méthodes correspondantes de la classe `DataHandler`.
3. **Exécution de l'action:** La méthode vérifie si l'action spécifiée existe dans la carte d'action, puis récupère et exécute cette méthode avec les arguments fournis (`**kwargs`).
4. **Gestion des exceptions:** Si l'action spécifiée n'est pas reconnue, une exception `ValueError` est levée.
5. **Retour du résultat:** Le résultat de l'action est retourné à l'appelant.

```
def data_handler_step(self, action, **kwargs):
    """
    Execute a data handler action.
```

```
        Args:
            action (str): The action to execute.
            **kwargs: Additional arguments for the action method.
        """
        self.data_handler = DataHandler(self.data_sources, self.step_results)

        action_map = {
            'get_data': self.data_handler.get_data,
            'set_data': self.data_handler.set_data,
            'delete_data': self.data_handler.delete_data,
            'search': self.data_handler.search,
            'create': self.data_handler.create,
            'evaluate': self.data_handler.evaluate,
        }

        if action not in action_map:
            raise ValueError(f"Unknown action {action}")

        return action_map[action](**kwargs)
```

Cette méthode offre une interface cohérente et flexible pour interagir avec des structures de données complexes, encapsulant les différentes actions dans une méthode unique et facilement configurable.

Étapes avec la Classe ApiHandler

La classe ApiHandler gère les requêtes API en utilisant les méthodes HTTP standard. Elle offre une interface simple pour interagir avec une API en spécifiant l'URL de base, l'endpoint, la méthode, les données, les paramètres, les en-têtes, et d'autres options.

Méthodes principales

Méthode	Description	Paramètres	Retour
`__init__`	Initialise l'ApiHandler avec l'URL de base et une option pour utiliser une session.	`base_url (str)` - URL de base de l'API. `use_session (bool)` - Si vrai, utilise une session pour les requêtes.	
`make_request`	Effectue une requête à l'API en utilisant la méthode, l'endpoint et les autres paramètres donnés.	`endpoint (str)`, `method (str)`, `data (dict)`, `params (dict)`, `headers (dict)`, `timeout (int)`, `encoding (str)`, `data_path (str)`, `retries (int)`	Les données de la réponse, ou `None` en cas d'échec.
`close`	Ferme la session, si une est ouverte.		

Exemples

`make_request`

Description: Effectue une requête à l'API en utilisant la méthode HTTP donnée, l'endpoint, et les autres paramètres. La méthode retourne les données de la réponse ou `None` en cas d'échec.

Configuration:

```
get_users:
  method: api_handler_step
  next_state: process_users
  parameters:
    api_url: "https://api.example.com"
    endpoint: "/users"
    method: "GET"
    params: {"limit": 10}
```

Paramètres

- `endpoint` (str): L'endpoint de l'API vers lequel la requête doit être faite. Cela sera ajouté à l'URL de base pour former l'URL complète de la requête.
- `method` (str): La méthode HTTP à utiliser pour la requête. Doit être l'une des suivantes: 'GET', 'POST', 'PUT', 'PATCH', 'DELETE'.
- `data` (dict): Les données à envoyer dans la requête (par exemple, le corps d'une requête POST).
- `params` (dict): Les paramètres à envoyer dans la requête (par exemple, les paramètres d'URL).
- `headers` (dict): Les en-têtes à envoyer avec la requête.
- `timeout` (int): Le délai maximal, en secondes, pour attendre une réponse de l'API.
- `encoding` (str): L'encodage à utiliser pour les données de la requête.
- `data_path` (str): Une expression JSONPath pour extraire des données spécifiques de la réponse.
- `retries` (int): Le nombre de fois que la requête doit être réessayée en cas d'échec.

Retour La méthode retourne les données de la réponse si la requête est réussie. Si la requête échoue, None est retourné. Si un chemin de données (`data_path`) est fourni, seul le sous-ensemble de données correspondant à cette expression JSONPath est retourné. Si la méthode HTTP est 'DELETE', le code d'état de la réponse est retourné au lieu des données.

Exceptions

- `ValueError`: Levée si une méthode HTTP invalide est spécifiée.
- `RequestException`: Peut être levée si une erreur se produit lors de l'exécution de la requête, comme un problème de connexion ou un code d'état non 200.

`close`

Description: Ferme la session ouverte par ApiHandler, si une existe.

Configuration:

```
close_session:
  method: api_handler_step
  next_state: finish
  parameters:
    action: close
```

Utilisation de la méthode `api_handler_step`

La méthode `api_handler_step` dans la classe principale joue un rôle essentiel dans la gestion des requêtes API. Elle permet d'exécuter une requête API en utilisant l'URL, l'endpoint, la méthode, et d'autres arguments.

Comment ça fonctionne

- **Définition de l'URL de l'API:** Si l'URL de l'API n'est pas fournie, elle est prise à partir de la configuration.
- **Initialisation de ApiHandler:** Une instance de la classe `ApiHandler` est créée avec l'URL de l'API et l'option de session.
- **Exécution de la requête:** La méthode `make_request` de l'instance `ApiHandler` est appelée avec l'endpoint, la méthode, et les autres arguments fournis (`**kwargs`).
- **Retour du résultat:** Le résultat de la requête est retourné à l'appelant.

```
api_request:
  method: api_handler_step
  next_state: process_response
  parameters:
    api_url: "https://api.example.com"
    endpoint: "/products"
    method: "POST"
    data: {"name": "New Product"}
```

```
def api_handler_step(self, api_url=None, endpoint='', method='GET',
use_session=False, **kwargs):
    if api_url is None:
        api_url = self.config['base_url']
    api_handler = ApiHandler(api_url, use_session)
    return api_handler.make_request(endpoint, method, **kwargs)
```

Cette méthode offre une interface cohérente et flexible pour interagir avec des APIs, encapsulant les différentes actions dans une méthode unique et facilement configurable.

Étapes avec la Classe UnifiedDataExtractor

La classe `UnifiedDataExtractor` fait partie d'un système d'extraction de données qui permet de traiter le contenu HTML brut et d'en extraire des informations spécifiques selon une configuration donnée. Cette classe

contient plusieurs méthodes pour extraire, formater, et gérer les données.

Méthodes principales

Méthode	Description	Paramètres	Retour
`perform_extraction`	Extrait les données du HTML brut en utilisant diverses techniques (JSON, XPath, regex, CSS, attributs, texte) selon la configuration.		Ajoute les données extraites à `self.extracted_data`.
`perform_formatting`	Formate les données extraites selon la configuration (conserver des clés spécifiques, ajouter des paires clé-valeur, rechercher des attributs, etc.).		Ajoute les données formatées à `self.formatted_data`.
`extract_and_format`	Effectue l'extraction et le formatage en appelant les méthodes `perform_extraction` et `perform_formatting` en séquence.		Retourne la liste des données formatées.

`perform_extraction`

Description: La méthode `perform_extraction` est responsable de l'extraction des données du contenu HTML brut. Elle utilise différentes techniques en fonction de la configuration. Vous pouvez choisir un élément HTML spécifique dans votre fichier de configuration, par exemple, une div avec une classe donnée, et spécifier ce que vous voulez trouver à l'intérieur de cette div en utilisant la clé `data_to_find`. Vous pouvez définir plusieurs types d'extraction comme JSON, XPath, regex, CSS, attributs, et texte.

Exemple de configuration YAML:

```
# Exemple de configuration pour l'extraction de données

selectors:                    # Début de la configuration de sélection
  - name: script               # Nom de la balise HTML à sélectionner (dans cet
exemple, une balise <script>)
    attrs: { type: application/json } # Attributs de la balise à sélectionner
(dans cet exemple, un attribut type avec la valeur "application/json")
    data_to_find:              # Début de la configuration pour spécifier les
données à trouver à l'intérieur de la balise sélectionnée
      url: { type: json, attribute:
props.pageProps.componentProps.adInfo.ad.friendlyUrl.url }  # Extraction JSON
      title: { type: json, attribute:
props.pageProps.componentProps.adInfo.ad.subject }        # Extraction JSON
      description: { type: json, attribute:
props.pageProps.componentProps.adInfo.ad.description } # Extraction JSON
      price: { type: xpath, xpath_expr: //span[@class="price"] }      # Extraction
XPath
      image: { type: regex, pattern: 'img src="(.*?)"' }              # Extraction
Regex
```

```
      header: { type: css, selector: 'h1.header' }                    # Extraction
CSS
      author: { type: attribute, attr_name: 'data-author' }           # Extraction
Attribut
      content: { type: text }                                         # Extraction
Texte
```

Dans cet exemple:

- **JSON:** Utilisé pour naviguer dans un objet JSON et extraire des valeurs spécifiques.
- **XPath:** Permet de naviguer dans le contenu HTML en utilisant des expressions XPath.
- **Regex:** Utilisé pour extraire des données en utilisant une expression régulière.
- **CSS:** Utilisé pour sélectionner des éléments en utilisant un sélecteur CSS.
- **Attribut:** Utilisé pour extraire la valeur d'un attribut HTML spécifique.
- **Texte:** Utilisé pour extraire le texte brut de l'élément HTML.

Cette configuration offre une grande flexibilité pour extraire divers types de données à partir du contenu HTML brut, en utilisant une variété de techniques. Il sert de guide pour écrire un fichier de configuration pour l'extraction de données et peut être adapté à vos besoins spécifiques.

`perform_formatting`

Description: La méthode `perform_formatting` prend les données extraites par `perform_extraction` et les formate selon les règles définies dans la configuration. Voici les différentes possibilités de formatage:

Exemple de configuration YAML:

```
# Exemple de configuration pour le formatage de données

formatting:                           # Début de la configuration de formatage
  keep: ['url', 'title', 'description'] # Clés spécifiques à conserver dans les
données formatées
  add:                                # Section pour ajouter des paires clé-valeur
spécifiques
    - key: 'state'                    # Clé à ajouter
      value: 'PROCESSED'              # Valeur correspondant à la clé ajoutée
  search:                             # Début de la configuration pour la
recherche et la conversion
    - method: 'search'                # Méthode de recherche (dans cet exemple,
une recherche basée sur un motif)
      attributes:                     # Attributs à rechercher
        - $key_name: 'author'         # Nom de la clé où sera sauvegardée la
donnée trouvée dans le résultat final
          pattern: '^Author:'         # Motif de recherche (expression régulière)
          return_key: 'name'          # Clé de retour spécifique (si applicable)
      convert: 'string'               # Conversion du résultat en type spécifié
(dans cet exemple, une chaîne)
    - method: 'jsonpath'              # Méthode d'extraction JSONPath
      jsonpath_expr: '$.author[*].name' # Expression JSONPath pour l'extraction
      $key_name: 'authors'            # Nom de la clé pour stocker les valeurs
```

```
extraites (résultat final)
```

Dans cet exemple:

- **Conserver des Clés Spécifiques (Keep):** La section 'keep' de la configuration permet de spécifier les clés à conserver dans les données formatées.
- **Ajouter des Paires Clé-Valeur (Add):** La section 'add' permet d'ajouter des paires clé-valeur spécifiques aux données.
- **Recherche et Conversion (Search):** La section 'search' permet de rechercher des attributs spécifiques et de les convertir dans un type donné. Il peut s'agir de recherches basées sur des motifs, des expressions JSONPath, ou d'autres méthodes de recherche.
- **Autres Méthodes de Formatage:** D'autres méthodes de formatage peuvent être implémentées selon les besoins, comme illustré par l'utilisation de JSONPath dans l'exemple.

La méthode `perform_formatting` permet de transformer et de formatter les données extraites en fonction des besoins spécifiques du projet. La configuration YAML fournit une manière flexible et extensible de définir les règles de formatage.

`extract_and_format`

Description: La méthode `extract_and_format` est la méthode principale à appeler pour commencer le processus d'extraction et de formatage des données. Elle combine les deux étapes précédentes (`perform_extraction` et `perform_formatting`) en une seule méthode, offrant une interface simple et cohérente. En premier lieu, elle appelle `perform_extraction` pour extraire les données brutes à partir du contenu HTML en se basant sur la configuration de sélection. Ensuite, elle appelle `perform_formatting` pour mettre en forme ces données selon la configuration de formatage.

Exemple de configuration YAML:

```yaml
# Exemple de configuration pour la méthode extract_and_format

extract_and_format:
  method: unified_extract_and_format
  next_state: update_property
  parameters:
    raw_data: '%%previous_result%%'
    config:
      selectors:                      # Début de la configuration de sélection
pour l'extraction
        - name: script                 # Nom de la balise HTML à sélectionner
        - attrs: { type: application/json } # Attributs de la balise à
sélectionner
        ...                            # Autres configurations de sélection
peuvent être ajoutées ici

      formatting:                     # Début de la configuration de formatage
        keep: ['url', 'title']        # Clés spécifiques à conserver dans les
données formatées
```

```
        add:
          - key: 'state'              # Ajout d'une paire clé-valeur
            value: 'PROCESSED'
        ...                            # Autres configurations de formatage
peuvent être ajoutées ici
```

Dans cet exemple:

- **Selectors:** La section 'selectors' de la configuration détaille comment les données doivent être extraites, y compris le nom de la balise HTML à sélectionner, les attributs à utiliser, etc.
- **Formatting:** La section 'formatting' de la configuration décrit comment les données extraites doivent être formatées. Elle peut inclure des directives pour conserver certaines clés, ajouter des paires clé-valeur spécifiques, et appliquer d'autres règles de formatage.

La méthode `extract_and_format` sert de point d'entrée principal pour le processus d'extraction et de formatage. Elle encapsule la logique nécessaire en une seule méthode, rendant le processus plus maniable et adaptable aux besoins spécifiques du projet. La configuration YAML offre une flexibilité permettant d'adapter facilement le processus aux exigences particulières.

Utilisation de la méthode `unified_extract_and_format`

La méthode `unified_extract_and_format` dans la classe principale joue un rôle vital dans l'extraction et la mise en forme des données à partir de contenu HTML brut. Elle utilise la classe `UnifiedDataExtractor` pour réaliser ces tâches.

Comment ça fonctionne

1. **Initialisation de UnifiedDataExtractor:** Une instance de la classe `UnifiedDataExtractor` est créée avec le contenu HTML brut et la configuration donnée (spécifiée dans le fichier YAML de configuration).
2. **Extraction et mise en forme des données:** La méthode `extract_and_format` de l'instance `UnifiedDataExtractor` est appelée, qui gère l'extraction et la mise en forme des données selon les règles définies dans la configuration.
3. **Retour des données formatées:** Les données extraites et mises en forme sont retournées à l'appelant sous forme d'une liste de dictionnaires.

```
def unified_extract_and_format(self, raw_data, config):
    """
    A method to extract data from raw HTML and then format it using the
UnifiedDataExtractor.

    Args:
        raw_data (str): The raw HTML content.
        config (dict): The configuration dictionary.

    Returns:
        list: A list of formatted data dictionaries.
    """
```

```
    extractor = UnifiedDataExtractor(raw_data, config)
    formatted_data = extractor.extract_and_format()

    return formatted_data
```

Cette méthode offre une interface cohérente et efficace pour extraire et formater des données à partir de contenu HTML brut, encapsulant les différentes actions dans une méthode unique et facilement configurable.

En résumé, la classe `UnifiedDataExtractor` offre une suite robuste et flexible de méthodes pour extraire et formater des données à partir de contenu HTML brut. Elle permet de définir des règles d'extraction et de formatage complexes via la configuration et offre un moyen efficace de gérer les données extraites pour diverses applications.

Étapes avec la Classe PageInteractor

La classe `PageInteractor` fournit des méthodes pour interagir avec des éléments de page web en utilisant Selenium. Elle permet de cliquer sur des éléments, d'entrer du texte, de défiler jusqu'à des éléments, de survoler des éléments, de défiler de manière spécifique et d'envoyer des touches clavier. Voici un aperçu des méthodes principales et un exemple de la façon dont elles peuvent être utilisées.

Méthodes principales

Méthode	Description	Paramètres	Retour
`click_element`	Clique sur un élément spécifié.	`by_method` - Méthode de localisation, `value` - Valeur pour localiser, `retries` (optionnel) - Nombre de tentatives.	`None`
`enter_text`	Entre du texte dans un élément spécifié.	`by_method`, `value`, `text` - Le texte à entrer, `retries` (optionnel).	`None`
`scroll_to_element`	Fait défiler jusqu'à un élément spécifié.	`by_method`, `value`, `retries` (optionnel).	`None`
`hover_over_element`	Survole un élément spécifié avec la souris.	`by_method`, `value`, `retries` (optionnel).	`None`
`scroll_by_amount`	Fait défiler de manière spécifique sur la page.	`x` (optionnel) - Défilement horizontal, `y` (optionnel) - Défilement vertical.	`None`
`send_key`	Envoie une touche clavier spécifique à un élément.	`by_method`, `value`, `key` - La touche à envoyer, `retries` (optionnel).	`None`
`interact_page`	Interagit avec la page en utilisant une liste d'interactions données. (voir exemple ci-dessous)	`interactions` - Une liste de dictionnaires représentant les interactions à effectuer.	`None`

Exemples

Voici comment ces méthodes peuvent être utilisées dans un flux de travail avec des exemples YAML :

Exemple 1 : Cliquer sur un bouton de connexion

```
connect:
  method: interact_page
  parameters:
    interactions:
      - {interaction: click, by_method: ID, value: 'login-button'}
  next_state: enter_credentials
```

Exemple 2 :Entrer du texte dans un champ de recherche et cliquer sur le bouton de recherche

```
search_item:
  method: interact_page
  parameters:
    interactions:
      - {interaction: enter_text, by_method: NAME, value: 'search-box', text:
'Laptop'}
      - {interaction: click, by_method: CLASS_NAME, value: 'search-button'}
  next_state: view_results
```

Exemple 3 : Faire défiler jusqu'à un élément spécifique sur la page

```
scroll_to_section:
  method: interact_page
  parameters:
    interactions:
      - {interaction: scroll, by_method: XPATH, value: '//*[@id="section-
target"]'}
  next_state: view_section
```

Utilisation de la méthode `interact_page`

La méthode `interact_page` dans la classe `PageInteractor` joue un rôle clé dans l'interaction avec la page web, permettant d'exécuter différentes actions d'interaction. Voici comment cela fonctionne :

1. **Lecture des Interactions :** La méthode prend une liste d'interactions en entrée. Chaque interaction est représentée par un dictionnaire contenant l'action à effectuer (par exemple, 'click', 'enter_text') et les détails nécessaires.

2. **Exécution des Interactions :** Chaque interaction est exécutée en séquence. Par exemple, si l'interaction est 'click', la méthode `click_element` est appelée avec les valeurs correspondantes.

3. **Gestion des Interactions Inconnues :** Si une interaction inconnue est rencontrée, un message d'erreur est affiché.

```
def interact_page(self, interactions):
    """ Interact with the page using the given interactions.
    Args:
        interactions (list): A list of dictionaries each representing an
interaction to be performed.
    """
    interactor = PageInteractor(self.driver)
    for interaction in interactions:
        by_method = getattr(By, interaction['by_method'])
        if interaction['interaction'] == 'click':
            interactor.click_element(by_method, interaction['value'])
        elif interaction['interaction'] == 'enter_text':
            text = interaction.get('text', '')
            interactor.enter_text(by_method, interaction['value'], text)
        elif interaction['interaction'] == 'scroll':
            interactor.scroll_to_element(by_method, interaction['value'])
        elif interaction['interaction'] == 'send_key':
            key = interaction.get('key', '')
            interactor.send_key(by_method, interaction['value'], key)
        else:
            print(f"Unknown interaction: {interaction['interaction']}")
```

Cette méthode offre une façon flexible et organisée d'interagir avec une page web en utilisant une liste d'interactions définies. Cela facilite la création de flux de travail complexes en définissant simplement une séquence d'interactions.