



# INTRODUCTION À SCRAPY

# QU'EST-CE QUE SCRAPY ?

Scrapy est un framework open-source écrit en Python. Il permet d'extraire des données de sites web de manière structurée. Utilisé pour le web scraping, il peut aussi crawler des sites web. Scrapy est conçu avec une architecture asynchrone.

# UTILISATION DE SCRAPY

- Extraire des données spécifiques d'un site.
- Stocker les données extraites dans divers formats.
- Suivre des liens pour naviguer dans un site.
- Gérer les sessions utilisateur et les cookies.
- Respecter les règles du fichier **robots .txt**.

# AVANTAGES DE SCRAPY

- Rapide et efficace.
- Extensible avec des middlewares et des extensions.
- Gestion aisée des requêtes parallèles.
- Supporte XPath et CSS pour la sélection des données.
- Intégration facile avec des bases de données et des fichiers.

# COMPOSANTS DE BASE DE SCRAPY

- **Spider**: Classe qui définit comment suivre les URL et extraire les données.
- **Item**: Conteneurs pour les données extraites.
- **Item Pipeline**: Traite les données extraites par les spiders.
- **Downloader**: Charge les pages web.
- **Engine**: Coordonne les composants précédents.

# FLUX DE TRAVAIL D'UNE ARAIGNÉE SCRAPY

1. Initialisation du Spider avec un ou plusieurs URL de départ.
2. Téléchargement des pages web.
3. Extraction des données et suivi des liens.
4. Traitement des données extraites.
5. Stockage des données dans le format souhaité.

# CAS D'UTILISATION COURANTS DE SCRAPY

- Surveillance des prix pour le commerce électronique.
- Collecte de données pour l'analyse de marché.
- Agrégation de contenu pour la curation de contenu.
- Acquisition de données pour l'apprentissage automatique.
- Surveillance de la conformité et de la présence en ligne.

# INSTALLATION DE SCRAPY

# PRÉREQUIS À L'INSTALLATION

- Système d'exploitation : Linux, Windows, macOS
- Python : Version 3.6 ou supérieure
- pip : Gestionnaire de paquets Python (dernière version recommandée)
- libxml2 et libxslt : Bibliothèques requises pour le traitement XML
- Installation des dépendances système spécifiques à chaque OS

# INSTALLATION AVEC PIP

Pour installer Scrapy, exécutez la commande suivante :

```
pip install scrapy
```

Cela téléchargera et installera Scrapy ainsi que ses dépendances.

# CRÉATION D'UN ENVIRONNEMENT VIRTUEL

1. Installer le paquet `virtualenv` si nécessaire :

```
pip install virtualenv
```

2. Créer un nouvel environnement virtuel :

```
virtualenv scrapy_env
```

3. Activer l'environnement virtuel :

- Linux/macOS: `source scrapy_env/bin/activate`
- Windows: `scrapy_env\Scripts\activate`

# VÉRIFICATION DE L'INSTALLATION

Pour vérifier que Scrapy est correctement installé, exécutez :

```
scrapy version
```

Si Scrapy est installé, cette commande affichera la version installée.

# ARCHITECTURE DE SCRAPY

# COMPOSANTS DE SCRAPY

- Engine: Coeur du système, gère le flux de données entre tous les composants.
- Scheduler: Enfile les requêtes et les délivre à l'Engine.
- Downloader: Récupère les pages Web et les renvoie à l'Engine.
- Spiders: Analysent les réponses et extraient les données.
- Item Pipeline: Traite les données extraites par les Spiders.
- Downloader Middlewares: Traite les requêtes et les réponses du Downloader.
- Spider Middlewares: Traite les entrées et sorties des Spiders.

# FLUX DE DONNÉES DANS SCRAPY

1. Engine envoie les requêtes au Scheduler.
2. Scheduler renvoie les requêtes à l'Engine.
3. Engine envoie les requêtes au Downloader via Downloader Middlewares.
4. Downloader renvoie les réponses à l'Engine.
5. Engine envoie les réponses aux Spiders via Spider Middlewares.
6. Spiders envoient les items extraits à l'Item Pipeline.
7. Item Pipeline traite et stocke les items.

# ENGINE DE SCRAPY

- Coordonne les autres composants de Scrapy.
- Gère le flux de données (requêtes et réponses).
- Envoie les requêtes depuis le Scheduler au Downloader.
- Distribue les réponses et les items aux Spiders et à l'Item Pipeline.
- Maintient l'état de l'exécution du crawling.

# SCHEDULER DE SCRAPY

- Organise l'ordre d'exécution des requêtes.
- Gère la politique de priorité des requêtes.
- Peut persister les requêtes pour stopper/reprendre le crawling.
- Communique directement avec l'Engine.

# DOWNLOADER DE SCRAPY

- Télécharge les pages Web en fonction des requêtes.
- Gère les politiques de téléchargement comme la rotation des agents utilisateur.
- Traite les erreurs de téléchargement et les redirections.
- Interagit avec Downloader Middlewares pour le traitement des requêtes/réponses.

# SPIDERS DE SCRAPY

- Classes définissant comment un site spécifique sera crawlé et extrait.
- Analyse les réponses téléchargées et extrait les données structurées.
- Génère des requêtes supplémentaires pour suivre les liens.
- Envoie les items extraits à l'Item Pipeline.

# ITEM PIPELINE DE SCRAPY

- Composé de plusieurs étapes de traitement des données extraites.
- Peut inclure la validation, le nettoyage et la persistance des données.
- Permet l'enrichissement des items et l'élimination des doublons.
- Connecté aux Spiders et à l'Engine.

# MIDDLEWARES DE SCRAPY

- Downloader Middlewares:
  - Situés entre l'Engine et le Downloader.
  - Peuvent modifier les requêtes et les réponses.
- Spider Middlewares:
  - Situés entre l'Engine et les Spiders.
  - Peuvent modifier les items et les réponses entrantes/sortantes.

# CRÉATION D'UN PROJET SCRAPY

# INSTALLATION DE SCRAPY

Pour installer Scrapy, ouvrez un terminal et exédez :

```
pip install scrapy
```

Assurez-vous que Python et pip sont déjà installés sur votre système.

# COMMANDE STARTPROJECT

Pour créer un nouveau projet Scrapy, utilisez la commande :

```
scrapy startproject nom_du_projet
```

Cela crée un répertoire **nom\_du\_projet** avec la structure nécessaire.

# STRUCTURE DU DOSSIER D'UN PROJET SCRAPY

La structure de base d'un projet Scrapy :

- nom\_projet/
  - spiders/ : Dossier pour vos spiders.
  - items.py : Définition des objets à extraire.
  - middlewares.py : Middlewares personnalisés.
  - pipelines.py : Traitement des items collectés.
  - settings.py : Configuration du projet.

# FICHIER SETTINGS.PY

Le fichier `settings.py` contient :

- Configuration du robot (user-agent, délai entre requêtes).
- Activation des extensions, middlewares et pipelines.
- Paramètres de déploiement.
- Autres configurations spécifiques au projet.

# FICHIER ITEMS.PY

Le fichier `items.py` définit :

- Les modèles d'items (objets à extraire).
- Utilisation de `scrapy.Item` et `scrapy.Field`.
- Exemple :

```
import scrapy

class MonItem(scrapy.Item):
    nom = scrapy.Field()
    prix = scrapy.Field()
```

# FICHIER MIDDLEWARES.PY

Le fichier `middlewares.py` :

- Permet de créer des middlewares personnalisés.
- Intercepte les requêtes/réponses.
- Peut modifier, rejeter ou enrichir les données.

# FICHIER PIPELINES.PY

Le fichier `pipelines.py`:

- Définit les pipelines de traitement des items.
- Peut nettoyer, valider ou stocker les items.
- Exemple de pipeline qui stocke dans une base de données.

# DÉFINITION D'UN SPIDER

# STRUCTURE DE BASE D'UN SPIDER

Un Spider Scrapy est une classe Python qui définit comment suivre les liens et extraire les données d'une page web. Voici les éléments clés :

- Nom du Spider
- URLs de départ
- Méthode de parsing

# IMPORTATION DES CLASSES NÉCESSAIRES

```
from scrapy import Spider
```

Pour créer un Spider, importez la classe `Spider` du module `scrapy`.

# CRÉATION D'UNE CLASSE SPIDER

```
class MonSpider(Spider):  
    # Le corps du spider sera défini ici
```

Créez une nouvelle classe héritant de `Spider` pour commencer à définir votre propre Spider.

# DÉFINITION DU NOM DU SPIDER

```
class MonSpider(Spider):  
    name = 'mon_spider'
```

Chaque Spider doit avoir un nom unique qui l'identifie.

# DÉFINITION DES URLs DE DÉPART (START\_URLS)

```
class MonSpider(Spider):
    start_urls = ['http://exemple.com']
```

`start_urls` est une liste d'URLs où le Spider commencera à crawler.

# MÉTHODE PARSE DU SPIDER

```
class MonSpider(Spider):
    def parse(self, response):
        # Extraction de données
```

La méthode `parse` est appelée avec la réponse du site web et contient la logique d'extraction.

# RÉPONSE ET RÉCUPÉRATION DE DONNÉES

```
def parse(self, response):
    title = response.css('h1::text').get()
```

Utilisez la réponse pour extraire des données, par exemple, le titre d'une page avec `css` selectors.

# GÉNÉRATION DE REQUÊTES SUIVANTES

```
def parse(self, response):
    for href in response.css('a::attr(href)'):
        yield response.follow(href, self.parse)
```

Générez des requêtes pour suivre les liens en utilisant `response.follow`.

# EXTRACTION DE DONNÉES (SELECTORS)

# SÉLECTEURS CSS

Les sélecteurs CSS permettent de cibler des éléments HTML spécifiques pour l'extraction de données.

Syntaxe courante : `tag, .class, #id, tag.class, tag#id`. Exemple : `response.css('title')` pour sélectionner la balise `<title>`.

# SÉLECTEURS XPATH

XPath est un langage de requête pour sélectionner des nœuds dans un document XML/HTML. Syntaxe : /, //, [@attrib='value'], /text(). Exemple : `response.xpath('//title')` pour sélectionner la balise `<title>`.

# UTILISATION DE LA MÉTHODE .SELECT()

La méthode `.select()` est obsolète dans Scrapy 1.0 et remplacée par `.xpath()`. Ancienne syntaxe : `response.select('//tag')`. Nouvelle syntaxe : `response.xpath('//tag')`.

# UTILISATION DE LA MÉTHODE .CSS()

La méthode `.css()` permet de sélectionner des éléments en utilisant les sélecteurs CSS. Syntaxe : `response.css('tag.class')`. Exemple : `response.css('a.button')` pour sélectionner des liens avec la classe `button`.

# EXTRACTION DE TEXTE AVEC `.EXTRACT()` ET `.EXTRACT_FIRST()`

`.extract()` retourne une liste de chaînes de caractères pour tous les éléments sélectionnés.

`.extract_first()` retourne la première chaîne de caractères correspondante. Exemple :

```
response.css('title::text').extract_first().
```

# EXTRACTION D'ATTRIBUTS

Pour extraire des attributs, utilisez la syntaxe `@attrib`. Exemple :

```
response.css('a::attr(href)').extract() pour obtenir les URLs de tous les liens <a>.
```

# CHAINAGE DES SÉLECTEURS

Les sélecteurs peuvent être chaînés pour affiner la sélection. Exemple :

```
response.css('div#content a::attr(href)').extract() pour les liens dans div#content.
```

# SÉLECTION D'ÉLÉMENTS PAR CLASSE ET IDENTIFIANT

Utilisez `.` pour les classes et `#` pour les identifiants. Exemple :

```
response.css('.product::text').extract() ou response.css('#main::text').extract()
```

# NAVIGATION DANS LE DOM

Navigation relative : > enfant direct, (espace) tous les enfants. Navigation absolue : ::text pour le texte, ::attr(attrib) pour un attribut. Exemple : response.css('div > p::text').extract() pour le texte des paragraphes enfants directs de div.

# UTILISATION DES EXPRESSIONS XPATH

# SYNTAXE DE BASE XPATH

XPath est un langage de requête pour sélectionner des nœuds dans un document XML. Scrapy utilise XPath pour naviguer dans les structures HTML. Syntaxe de base :

```
response.xpath('xpath_expression')
```

# SÉLECTION D'ÉLÉMENTS

Pour sélectionner des éléments avec XPath :

- Sélectionner tous les éléments : `//tag`
- Sélectionner éléments spécifiques : `//tag[@attr='value']`

# UTILISATION DES PRÉDICATS

Les prédictats filtrent les nœuds sélectionnés :

```
//tag[predicate]
```

Exemples :

- Sélection par index : `//tag[index]`
- Sélection par attribut : `//tag[@attr='value']`

# SÉLECTION D'ATTRIBUTS

Pour obtenir la valeur d'un attribut :

```
//tag/@attr
```

Exemple :

- Sélectionner l'attribut href des liens : //a/@href

# NAVIGATION DANS L'ARBORESCENCE

- Parent : ..
- Enfants : /
- Descendants : //
- Frères suivants : **following-sibling::**
- Frères précédents : **preceding-sibling::**

# FONCTIONS XPATH COURANTES

- `text()` : Sélectionne le texte des éléments.
- `normalize-space()` : Supprime les espaces superflus.
- `contains(@attr, 'string')` : Vérifie si l'attribut contient une chaîne.

# COMBINAISON D'EXPRESSIONS XPATH

Expressions XPath peuvent être combinées :

- Union : |
- Chemins conditionnels : //a[condition]/b

# ESPACES DE NOMS ET XPATH

Pour gérer les espaces de noms XML dans XPath :

- Utiliser `local-name()` pour ignorer l'espace de noms.
- Déclarer un préfixe d'espace de noms et l'utiliser dans la requête.

# UTILISATION DES SÉLECTEURS CSS

# COMPRENDRE LES SÉLECTEURS CSS

Les sélecteurs CSS permettent de cibler des éléments HTML. Utilisés pour extraire des données spécifiques d'une page web. Scrapy utilise les sélecteurs CSS pour parcourir le DOM. Syntaxe similaire à celle utilisée en CSS pour le style des pages web.

# UTILISATION DE LA MÉTHODE .CSS()

```
response.css('selector')
```

Permet de sélectionner des éléments HTML via leur sélecteur CSS. Retourne une liste de sélecteurs correspondant aux éléments trouvés. Méthode de la classe **Selector** de Scrapy.

# EXTRACTION DE DONNÉES AVEC .GET() ET .GETALL()

```
response.css('selector').get() # Retourne le premier élément trouvé.  
response.css('selector').getall() # Retourne tous les éléments trouvés.
```

.get() et .getall() extraient les données des éléments sélectionnés. .get() est équivalent à .extract\_first() dans les versions antérieures.

# SÉLECTIONNER DES ATTRIBUTS AVEC LES SÉLECTEURS CSS

```
response.css('a::attr(href)').get()
```

Utilise `::attr(attribut)` pour extraire la valeur d'un attribut spécifique. Permet d'obtenir des liens, des chemins d'images, etc.

# UTILISATION DES PSEUDO-CLASSES CSS

```
response.css('a::text').get()  
response.css('li:first-child').get()
```

Les pseudo-classes (`::text`, `:first-child`, etc.) ciblent des éléments spéciaux. `::text` extrait le texte intérieur d'un élément.

# COMBINAISON DE SÉLECTEURS CSS POUR DES REQUÊTES COMPLEXES

```
response.css('div.content > p::text').getall()
```

Combinez les sélecteurs pour affiner la sélection. `>` sélectionne les enfants directs, `(espace)` sélectionne tous les descendants.

# GESTION DES ITEMS

# DÉFINITION D'UN ITEM

Les Items dans Scrapy sont des conteneurs pour collecter les données scrapées. Ils fonctionnent comme des dictionnaires Python simples. Les Items définissent les champs pour stocker les données spécifiques à scraper.

# CRÉATION DE CLASSES ITEM

Pour créer un Item dans Scrapy :

```
import scrapy

class MonItem(scrapy.Item):
    # déclaration des champs ici
```

Chaque champ de l'Item représente une donnée que vous souhaitez extraire.

# CHAMPS D'ITEM

Les champs d'Item sont déclarés dans les classes Item :

```
class MonItem(scrapy.Item):
    titre = scrapy.Field()
    prix = scrapy.Field()
```

Ces champs sont utilisés pour définir la structure des données scrapées.

# UTILISATION DE FIELD POUR LES ITEMS

`scrapy.Field()` permet de spécifier les métadonnées pour chaque champ :

```
class MonItem(scrapy.Item):
    titre = scrapy.Field(serializer=str)
    prix = scrapy.Field(serializer=int)
```

Les métadonnées peuvent inclure des validateurs, des sérialiseurs, etc.

# EXTRACTION ET ASSIGNATION DES DONNÉES AUX ITEMS

Extraction des données avec les sélecteurs :

```
def parse(self, response):
    item = MonItem()
    item['titre'] = response.css('h1::text').get()
    item['prix'] = response.css('.prix::text').get()
    return item
```

Assignez les données extraites aux champs correspondants de l'Item.

# EXPORTATION DES ITEMS VERS DES FICHIERS

Scrapy peut exporter des Items dans différents formats :

- JSON
- CSV
- XML

Utilisez la commande suivante pour exporter :

```
scrapy crawl monspider -o resultats.json
```

Cela génère un fichier `resultats.json` contenant les données des Items.

# PIPELINE DE TRAITEMENT DES DONNÉES

# STRUCTURE D'UN PIPELINE

- Les pipelines traitent les items collectés par le spider
- Composés de plusieurs méthodes spéciales
- Peuvent effectuer des tâches comme :
  - Nettoyage des données
  - Validation
  - Stockage dans une base de données

# ACTIVATION ET ORDRE DES PIPELINES

- Définis dans le fichier `settings.py`
- Activés par la variable `ITEM_PIPELINES`
- L'ordre est déterminé par des valeurs numériques :
  - Plus la valeur est basse, plus tôt le pipeline est exécuté

```
ITEM_PIPELINES = {  
    'myproject.pipelines.ExamplePipeline': 300,  
}
```

# MÉTHODES DU PIPELINE DE TRAITEMENT

- `open_spider(self, spider)`: appelée quand le spider est ouvert
- `close_spider(self, spider)`: appelée quand le spider est fermé
- `process_item(self, item, spider)`: appelée pour chaque item

# STOCKAGE DES DONNÉES EXTRAITES

- `process_item` peut être utilisé pour stocker des données
- Exemples de stockage :
  - Bases de données (MySQL, MongoDB)
  - Fichiers (CSV, JSON, XML)

# NETTOYAGE DES DONNÉES

- Nettoyage effectué dans `process_item`
- Opérations courantes :
  - Suppression des espaces blancs
  - Conversion de types de données
  - Normalisation de texte

# VALIDATION DES DONNÉES

- Validation effectuée dans `process_item`
- Assure la qualité des données
- Utilisation de librairies comme `jsonschema` ou `cerberus`

# GESTION DES ERREURS DANS LE PIPELINE

- Gérer les exceptions dans `process_item`
- Utiliser `try` et `except` pour traiter les erreurs
- Logger les erreurs avec le module `logging`

```
try:  
    # Traitement des données  
except Exception as e:  
    self.logger.error(f"Erreur lors du traitement: {e}")
```

# MIDDLEWARE DE SCRAPY

# FONCTIONNEMENT DES MIDDLEWARES

Les Middlewares dans Scrapy sont des composants qui permettent de personnaliser le traitement des requêtes et des réponses. Ils s'insèrent dans le flux de traitement des requêtes et réponses, agissant comme des filtres. Les Middlewares peuvent modifier, ajouter ou supprimer des requêtes ou des réponses.

# MIDDLEWARES INTÉGRÉS

Scrapy offre plusieurs Middlewares intégrés pour des tâches communes :

- Gestion des cookies
- Respect des règles du robots.txt
- Gestion des redirections
- Gestion des erreurs HTTP
- Compression des requêtes et réponses

# CRÉER UN MIDDLEWARE PERSONNALISÉ

Pour créer un Middleware personnalisé :

1. Définir une classe Python
2. Implémenter les méthodes `process_request` et/ou `process_response`
3. Ajouter le Middleware à `settings.py`
4. Activer le Middleware dans la configuration

# ACTIVER ET DÉSACTIVER LES MIDDLEWARES

Pour activer un Middleware :

1. Ajouter le chemin de la classe à `DOWNLOADER_MIDDLEWARES` ou `SPIDER_MIDDLEWARES` dans `settings.py`
2. Assigner un ordre de priorité

Pour désactiver un Middleware :

1. Commenter ou supprimer la ligne correspondante dans `settings.py`

# ORDRE DES MIDDLEWARES

L'ordre des Middlewares est déterminé par des nombres entiers dans `settings.py`. Les Middlewares avec un nombre plus petit sont exécutés en premier. L'ordre influence comment les requêtes et réponses sont traitées à travers la chaîne de Middlewares.

# MIDDLEWARE POUR LE TÉLÉCHARGEMENT (DOWNLOADER MIDDLEWARE)

Downloader Middleware :

- Intercepte les requêtes avant qu'elles ne soient effectuées
- Intercepte les réponses avant qu'elles ne soient transmises aux Spiders
- Peut modifier, rejeter ou ajouter des requêtes et réponses

# MIDDLEWARE POUR LE SPIDER (SPIDER MIDDLEWARE)

Spider Middleware :

- Intercepte les réponses après qu'elles soient téléchargées mais avant le parsing par le Spider
- Intercepte les résultats (items et requêtes) du Spider avant qu'ils ne soient transmis au moteur
- Peut modifier, rejeter ou ajouter des items et des requêtes

# CONFIGURATION DE SCRAPY

# INSTALLATION DE SCRAPY

Pour installer Scrapy, utilisez pip :

```
pip install scrapy
```

Assurez-vous que pip est installé et à jour. Scrapy nécessite Python version 3.6 ou supérieure.

# STRUCTURE D'UN PROJET SCRAPY

```
monprojet/
    scrapy.cfg
    monprojet/
        __init__.py
        items.py
        middlewares.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
            mon_spider.py
```

Chaque projet Scrapy a cette structure de base.

# FICHIER SETTINGS.PY

Le fichier `settings.py` gère la configuration :

- Définit les paramètres par défaut.
- Contrôle les middlewares et les pipelines.
- Configure les extensions et le `robot.txt`.

# PERSONNALISATION DES PARAMÈTRES

Pour personnaliser les paramètres :

```
# Dans settings.py
BOT_NAME = 'mon_bot'
LOG_LEVEL = 'ERROR'
ITEM_PIPELINES = {
    'monprojet.pipelines.ExemplePipeline': 300,
}
```

Priorité des pipelines de 0 à 1000.

# GESTION DES EXTENSIONS

Activer ou désactiver les extensions dans `settings.py`:

```
EXTENSIONS = {  
    'scrapy.extensions.telnet.TelnetConsole': None,  
}
```

`None` désactive l'extension.

# CONFIGURATION DU USER AGENT

Modifier le User Agent :

```
# Dans settings.py
USER_AGENT = 'monprojet (+http://www.mon-site.com)'
```

Simulez un navigateur spécifique ou un robot d'indexation.

# CONFIGURATION DU ROBOT.TXT

Pour respecter les règles robot.txt :

```
# Dans settings.py
ROBOTSTXT_OBEY = True
```

**True** pour activer, **False** pour ignorer.

# LIMITATION DU TAUX DE REQUÊTES

Configurer le délai entre les requêtes :

```
# Dans settings.py
DOWNLOAD_DELAY = 2.5
```

Le délai est en secondes.

# DÉPLOIEMENT D'UN SPIDER SCRAPY

# INSTALLATION DE SCRAPYD

Pour installer Scrapyd, exécutez la commande suivante :

```
pip install scrapyd
```

Scrapyd est un service pour déployer et exécuter des spiders Scrapy. Il permet de gérer facilement vos spiders.

# CONFIGURATION DE SCRAPYD

Fichier de configuration par défaut : `scrapyd.conf`

```
[scrapyd]
bind_address = 127.0.0.1
http_port = 6800
```

Modifiez ce fichier pour personnaliser les paramètres de Scrapyd.

# DÉPLOIEMENT AVEC SCRAPYD-CLIENT

Pour déployer un spider avec scrapyd-client :

1. Installez scrapyd-client :

```
pip install scrapyd-client
```

2. Utilisez la commande **deploy** :

```
scrapyd-deploy <target> -p <project>
```

# GESTION DES VERSIONS DE SPIDER

Chaque déploiement est associé à une version.

Utilisez l'option `-v <version>` avec `scrapyd-deploy` pour spécifier une version.

```
scrapyd-deploy <target> -p <project> -v <version>
```

Les versions permettent de revenir à une version antérieure si nécessaire.

# PLANIFICATION DE L'EXÉCUTION DU SPIDER

Pour planifier un spider :

1. Envoyez une requête POST à Scrapyd :

```
curl http://<host>:6800/schedule.json -d project=<project> -d spider=<spider>
```

2. Spécifiez l'intervalle d'exécution avec `add_job_schedule` dans `scrapyd.conf`.

# MONITORING DES SPIDERS DÉPLOYÉS

Pour surveiller vos spiders :

1. Accédez à `http://<host>:6800/` pour le dashboard Scrapyd.
2. Utilisez l'API Scrapyd pour obtenir des informations :

```
curl http://<host>:6800/listjobs.json?project=<project>
```

Le monitoring permet de suivre l'état et la progression des spiders.

# LOGGING AVEC SCRAPY

# CONFIGURATION DU LOGGING

Pour configurer le logging dans Scrapy, modifiez les paramètres `LOG_LEVEL` et `LOG_FILE` dans le fichier `settings.py`.

```
LOG_LEVEL = 'INFO' # Niveaux: CRITICAL, ERROR, WARNING, INFO, DEBUG
LOG_FILE = 'scrapy_log.txt' # Nom du fichier de log
```

# NIVEAUX DE LOG

Les niveaux de log déterminent l'importance des messages à enregistrer.

- **CRITICAL** : erreurs graves pouvant entraîner l'arrêt du programme
- **ERROR** : erreurs graves qui ne nécessitent pas l'arrêt du programme
- **WARNING** : avertissements, situations non critiques nécessitant une attention
- **INFO** : confirmations que les choses fonctionnent comme prévu
- **DEBUG** : informations détaillées, utiles pour le débogage

# CAPTURE ET AFFICHAGE DES LOGS

Capturer les logs dans la console :

```
import scrapy
from scrapy.utils.log import configure_logging
configure_logging(install_root_handler=False)
```

Affichage des logs dans la console :

```
import logging
logging.info('Information message')
logging.error('Error message')
```

# FICHIERS DE LOG

Pour enregistrer les logs dans un fichier :

```
import logging

logging.basicConfig(
    filename='scrapy_log.txt',
    format='%(levelname)s: %(message)s',
    level=logging.INFO
)
```

# FILTRAGE DES LOGS

Pour ignorer les logs en dessous d'un certain niveau :

```
LOG_LEVEL = 'WARNING'
```

Pour filtrer les logs par composant :

```
LOGGING = {
    'loggers': {
        'scrapy': {'level': 'INFO'},
        'scrapy.middleware': {'level': 'WARNING'},
    }
}
```

# GESTION DES ERREURS ET DEBUGGING

# COMPRENDRE LES EXCEPTIONS DANS SCRAPY

Les exceptions dans Scrapy sont des erreurs qui surviennent pendant l'exécution d'un spider. Elles peuvent être causées par des problèmes de réseau, des erreurs de code ou des données manquantes. Scrapy fournit des exceptions intégrées pour gérer ces problèmes. Exemples : `CloseSpider`, `DropItem`, `IgnoreRequest`.

# UTILISATION DE LA CONSOLE SCRAPY SHELL POUR LE DÉBOGAGE

La console Scrapy Shell est un outil interactif pour tester et déboguer votre spider. Elle permet d'exécuter des commandes Scrapy dans un environnement Python. Utilisez `scrapy shell 'url'` pour démarrer une session avec une URL spécifique. Testez les sélecteurs et les requêtes sans exécuter tout le spider.

# INTERPRÉTATION DES ERREURS COURANTES

Type d'erreur	Description
Erreurs de connexion	Problèmes de réseau ou URL incorrecte
Sélecteurs incorrects	XPath ou CSS ne trouvent pas l'élément cible
Erreurs de type de données	Mauvais type de donnée extrait ou manipulé
Erreurs de middleware	Problèmes dans les middlewares personnalisés
Erreurs de pipeline	Soucis lors de l'enregistrement des données

# GESTION DES ERREURS DANS LES SPIDERS

Pour gérer les erreurs dans les spiders :

- Utilisez des blocs `try` et `except` pour capturer et traiter les exceptions.
- Loggez les erreurs avec les niveaux appropriés (ERROR, WARNING).
- Utilisez `raise` pour générer des exceptions personnalisées si nécessaire.

# UTILISATION DES MIDDLEWARES POUR GÉRER LES ERREURS

Les middlewares sont des composants qui s'exécutent avant ou après les spiders. Ils peuvent être utilisés pour gérer les erreurs de manière centralisée. Créez un middleware personnalisé pour traiter les réponses et les exceptions. Configurez-le dans `settings.py` sous `DOWNLOADER_MIDDLEWARES`.

# CONFIGURATION DES PARAMÈTRES DE DÉBOGAGE

Configurez les paramètres de débogage dans `settings.py` :

- `LOG_LEVEL` pour définir le niveau de log (DEBUG, INFO, WARNING, ERROR).
- `LOG_FILE` pour spécifier un fichier où enregistrer les logs.
- `LOG_ENABLED` pour activer ou désactiver le logging.

# UTILISATION DES SIGNAUX POUR SURVEILLER LES ERREURS

Scrapy fournit des signaux que vous pouvez écouter pour surveiller les événements. Utilisez des signaux pour exécuter du code en réponse à certaines actions (comme des erreurs). Exemples de signaux : `spider_error`, `item_dropped`. Connectez des fonctions aux signaux via `dispatcher.connect()`.

# ANALYSE DES LOGS POUR IDENTIFIER LES PROBLÈMES

Les logs contiennent des informations détaillées sur l'exécution du spider. Recherchez des patterns d'erreurs pour identifier les problèmes récurrents. Utilisez des outils comme grep ou des analyseurs de logs pour filtrer les informations. Les logs vous aident à comprendre le comportement de votre spider et à optimiser son fonctionnement.

# RESPECT DES RÈGLES DU ROBOTS.TXT

# COMPRÉHENSION DU FICHIER ROBOTS.TXT

Le fichier **robots .txt** est un fichier texte placé à la racine d'un site web. Il indique aux robots d'indexation (web crawlers) les sections du site à ne pas explorer. La syntaxe **Disallow:** spécifie les chemins d'accès interdits aux robots. La directive **User-agent:** définit pour quel robot les règles s'appliquent. Un **robots .txt** peut contenir plusieurs groupes de règles pour différents robots.

# UTILISATION DE LA CLASSE ROBOTFILEPARSER

```
from urllib.robotparser import RobotFileParser
```

- Créer une instance : rp = RobotFileParser()
- Charger le fichier robots.txt : rp.set\_url('http://example.com/robots.txt')
- Lire le fichier : rp.read()
- Vérifier l'accès : rp.can\_fetch('\*', '/chemin/')

# CONFIGURATION DE SCRAPY POUR RESPECTER ROBOTS.TXT

Dans Scrapy, respecter `robots.txt` est une question de configuration :

- Définir les paramètres dans `settings.py`.
- Utiliser `ROBOTSTXT_OBEY` pour activer ou désactiver le respect de `robots.txt`.
- Scrapy utilise `RobotFileParser` pour lire et respecter les règles.

# ACTIVATION DE L'OPTION ROBOTSTXT\_OBEY

Pour activer le respect de `robots.txt` dans Scrapy :

```
# Dans settings.py
ROBOTSTXT_OBEY = True
```

- Lorsque `ROBOTSTXT_OBEY` est à `True`, Scrapy vérifie `robots.txt` avant de scraper.
- Si à `False`, Scrapy ignorera les règles de `robots.txt`.

# CONSÉQUENCES DU NON-RESPECT DE ROBOTS.TXT

Ignorer robots.txt peut entraîner :

- Un accès bloqué au site par l'administrateur.
- Une surcharge des serveurs du site, nuisant à son fonctionnement.
- Des problèmes juridiques liés à l'accès non autorisé aux données.

# GESTION DES RESTRICTIONS D'ACCÈS PAR ROBOTS.TXT

Scrapy gère automatiquement les restrictions si `ROBOTSTXT_OBEY` est à `True`.

- Les requêtes vers des chemins interdits sont évitées.
- Les spiders sont conçus pour être respectueux et éviter le bannissement.
- Il est important de vérifier manuellement `robots.txt` pour des règles complexes.

# PLANIFICATION ET EXÉCUTION DES SPIDERS

# INSTALLATION DE SCRAPY

Pour installer Scrapy, utilisez pip :

```
pip install scrapy
```

Assurez-vous que Python et pip sont déjà installés sur votre système.

# CRÉATION D'UN PROJET SCRAPY

Pour créer un nouveau projet Scrapy :

```
scrapy startproject nom_du_projet
```

Cela crée un dossier **nom\_du\_projet** avec la structure nécessaire.

# GÉNÉRATION D'UN SPIDER

Pour générer un spider dans votre projet :

```
scrapy genspider nom_du_spider domaine.com
```

Remplacez **nom\_du\_spider** et **domaine.com** par vos valeurs.

# CONFIGURATION DU SPIDER

Dans le fichier `settings.py` :

- Configurez les paramètres du spider.
- Définissez `USER_AGENT` pour respecter les règles du site.
- Ajustez les paramètres de concurrence et de délai si nécessaire.

# DÉFINITION DES RÈGLES DE SUIVI DES LIENS

Dans votre spider, définissez les règles :

```
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor

class MonSpider(CrawlSpider):
    name = 'mon_spider'
    allowed_domains = ['domaine.com']
    start_urls = ['http://www.domaine.com']
    rules = (
        Rule(LinkExtractor(allow=()), callback='parse_item', follow=True),
    )
```

# EXTRACTION DES DONNÉES

Définissez la méthode `parse` ou `parse_item`:

```
def parse_item(self, response):
    yield {
        'titre': response.css('h1::text').get(),
        'url': response.url,
    }
```

# UTILISATION DES SÉLECTEURS

Utilisez `css` ou `xpath` pour sélectionner les éléments :

```
response.css('div.class > a::attr(href)').getall()
response.xpath('//div[@class="class"]/a/@href').extract()
```

# GESTION DES REQUÊTES

Utilisez Request pour suivre les liens :

```
from scrapy import Request

def parse(self, response):
    for href in response.css('a::attr(href)'):
        yield response.follow(href, self.parse_item)
```

# TRAITEMENT DES ÉLÉMENTS EXTRAITS

Utilisez les **Items** et **Item Loaders** pour un traitement cohérent :

```
from scrapy.loader import ItemLoader
from scrapy.loader.processors import TakeFirst

class MonItem(scrapy.Item):
    titre = scrapy.Field(output_processor=TakeFirst())
```

# STOCKAGE DES DONNÉES

Choisissez un format de stockage dans `settings.py`:

```
FEED_FORMAT = 'json'  
FEED_URI = 'resultat.json'
```

# EXÉCUTION DU SPIDER

Pour exécuter votre spider :

```
scrapy crawl nom_du_spider
```

Remplacez **nom\_du\_spider** par le nom de votre spider.

# SURVEILLANCE ET DÉBOGAGE DU SPIDER

Utilisez les commandes suivantes pour déboguer :

```
scrapy shell 'http://www.domaine.com'  
scrapy crawl nom_du_spider -o sortie.json -t json
```

# PLANIFICATION PÉRIODIQUE DES SPIDERS (AVEC CRON OU AUTRE)

Pour une exécution périodique, utilisez cron :

```
0 * * * * cd chemin_du_projet && scrapy crawl nom_du_spider
```

Adaptez la commande à votre environnement et fréquence désirée.