Formation ETL avec Python

m2iformation.fr

Formation ETL avec Python

Extraction, Transformation et Chargement de données

Objectifs de la formation

Compétences visées

- Maîtriser l'extraction de données depuis diverses sources (CSV, Excel, API)
- Transformer et nettoyer les données avec Python et Pandas
- Construire des pipelines ETL robustes et automatisés
- Gérer les erreurs et la qualité des données

Public

Concepteurs et développeurs data, data engineers, data analysts

Prérequis

Connaissances de base en Python (variables, fonctions, structures de contrôle)

Introduction à l'ETL

Qu'est-ce que l'ETL ?

ETL = Extract, Transform, Load

Extract (Extraction)

- Récupérer les données depuis différentes sources
- Sources : fichiers, bases de données, API, web scraping...

Transform (Transformation)

- Nettoyer les données (valeurs manquantes, doublons)
- Convertir les formats (dates, types de données)
- Enrichir les données (calculs, agrégations)
- Valider la qualité

Load (Chargement)

- Stocker les données transformées
- Destinations : bases de données, data warehouses, fichiers...

Pourquoi l'ETL est essentiel

Cas d'usage en entreprise

1. **Consolidation de données**
   - Agréger des données de plusieurs systèmes
   - Exemple : Combiner ventes online + magasins physiques
2. **Migration de données**
   - Passage d'un système à un autre
   - Transformation de formats
3. **Business Intelligence**
   - Alimenter des data warehouses
   - Préparer les données pour l'analyse
4. **Automatisation**
   - Processus réguliers (quotidiens, horaires)
   - Réduction des tâches manuelles

Fichiers CSV et Excel

Fichiers CSV - Rappels

CSV = Comma-Separated Values

Format texte simple pour les données tabulaires.

Structure

```
nom,prenom,age,ville
Dupont,Jean,35,Paris
Martin,Marie,28,Lyon
Durant,Pierre,42,Marseille
```

Caractéristiques

- Séparateur : virgule (ou point-virgule, tabulation)
- Première ligne : souvent les en-têtes
- Format léger et universel
- Pas de formules ni de formatage

Lecture CSV - Python natif

Module csv de Python

```python
import csv

# Lecture d'un fichier CSV
with open('data.csv', 'r', encoding='utf-8') as file:
    reader = csv.reader(file)

    # Lire l'en-tête
    headers = next(reader)
    print(f"Colonnes : {headers}")

    # Lire les données
    for row in reader:
        print(row)

# Lecture avec DictReader (plus pratique)
with open('data.csv', 'r', encoding='utf-8') as file:
    reader = csv.DictReader(file)

    for row in reader:
        print(f"{row['nom']} - {row['age']} ans")
```

Écriture CSV - Python natif

```python
import csv

# Données à écrire
data = [
    ['nom', 'prenom', 'age', 'ville'],
    ['Dupont', 'Jean', 35, 'Paris'],
    ['Martin', 'Marie', 28, 'Lyon'],
    ['Durant', 'Pierre', 42, 'Marseille']
]

# Écriture
with open('output.csv', 'w', newline='', encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerows(data)

# Avec DictWriter
with open('output2.csv', 'w', newline='', encoding='utf-8') as file:
    fieldnames = ['nom', 'prenom', 'age', 'ville']
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'nom': 'Dupont', 'prenom': 'Jean', 'age': 35, 'ville': 'Paris'})
```

Pandas - Introduction

Bibliothèque incontournable pour la data

Pandas fournit des structures de données puissantes :

- **Series** : Tableau 1D (comme une colonne)
- **DataFrame** : Tableau 2D (comme une feuille Excel)

Avantages

- Manipulation de données intuitive
- Opérations vectorisées (rapides)
- Gestion automatique des types
- Nombreuses fonctions d'analyse
- Intégration avec NumPy, Matplotlib...

Import

```
import pandas as pd
```

DataFrame Pandas

Création d'un DataFrame

```
import pandas as pd

# Depuis un dictionnaire
data = {
    'nom': ['Dupont', 'Martin', 'Durant'],
    'prenom': ['Jean', 'Marie', 'Pierre'],
    'age': [35, 28, 42],
    'ville': ['Paris', 'Lyon', 'Marseille']
}
df = pd.DataFrame(data)
print(df)
```

Résultat

```
      nom  prenom  age      ville
0  Dupont    Jean   35      Paris
1  Martin   Marie   28       Lyon
2  Durant  Pierre   42  Marseille
```

Lecture CSV avec Pandas

Fonction read_csv()

```
import pandas as pd

# Lecture basique
df = pd.read_csv('data.csv')

# Avec options
df = pd.read_csv(
    'data.csv',
    sep=',',               # Séparateur (auto-détecté par défaut)
    encoding='utf-8',      # Encodage
    na_values=['NA', ''], # Valeurs considérées comme manquantes
    parse_dates=['date'], # Colonnes à parser comme dates
    dtype={'age': int},    # Types de données
    nrows=1000           # Lire seulement les 1000 premières lignes
)

# Afficher les premières lignes
print(df.head())

# Informations sur le DataFrame
print(df.info())
```

Exploration de données Pandas

```
# Dimensions
print(df.shape)  # (lignes, colonnes)

# Premières et dernières lignes
print(df.head(10))  # 10 premières
print(df.tail(5))   # 5 dernières

# Informations sur les colonnes
print(df.info())

# Statistiques descriptives
print(df.describe())

# Noms des colonnes
print(df.columns)

# Types de données
print(df.dtypes)

# Valeurs manquantes
print(df.isnull().sum())

# Valeurs uniques
print(df['ville'].unique())
print(df['ville'].nunique())  # Nombre de valeurs uniques
```

Sélection de données

```
# Sélectionner une colonne
ages = df['age']
villes = df.ville  # Notation pointée (si pas d'espaces dans le nom)

# Sélectionner plusieurs colonnes
subset = df[['nom', 'age']]

# Sélectionner des lignes par index
premiere_ligne = df.iloc[0]  # Première ligne
lignes_0_a_4 = df.iloc[0:5]  # 5 premières lignes

# Sélectionner par label
df_with_index = df.set_index('nom')
jean = df_with_index.loc['Dupont']

# Filtrage avec conditions
adultes = df[df['age'] >= 18]
parisiens = df[df['ville'] == 'Paris']
jeunes_parisiens = df[(df['age'] < 30) & (df['ville'] == 'Paris')]

# Plusieurs conditions
condition = (df['age'] > 25) & (df['age'] < 40) & (df['ville'].isin(['Paris', 'Lyon']))
resultat = df[condition]
```

Modification de données

```python
# Ajouter une colonne
df['pays'] = 'France'
df['age_plus_10'] = df['age'] + 10

# Modifier une colonne
df['age'] = df['age'] + 1

# Renommer des colonnes
df = df.rename(columns={'nom': 'nom_famille', 'prenom': 'prenom'})

# Supprimer des colonnes
df = df.drop(columns=['age_plus_10'])
df = df.drop(['colonne1', 'colonne2'], axis=1)

# Supprimer des lignes
df = df.drop([0, 1])  # Par index
df = df[df['age'] > 20]  # Par condition

# Remplacer des valeurs
df['ville'] = df['ville'].replace('Paris', 'Paris 75')
df = df.replace({'ville': {'Paris': 'Paris 75', 'Lyon': 'Lyon 69'}})
```

Gestion des valeurs manquantes

```
# Détecter les valeurs manquantes
print(df.isnull())
print(df.isnull().sum())  # Nombre par colonne

# Supprimer les lignes avec valeurs manquantes
df_clean = df.dropna()  # Toute ligne avec au moins un NaN
df_clean = df.dropna(subset=['age'])  # Seulement si 'age' est NaN

# Supprimer les colonnes avec trop de valeurs manquantes
df_clean = df.dropna(axis=1, thresh=len(df)*0.8)  # Garde si 80% de données

# Remplir les valeurs manquantes
df['age'].fillna(0, inplace=True)  # Remplacer par 0
df['age'].fillna(df['age'].mean(), inplace=True)  # Par la moyenne
df['ville'].fillna('Inconnue', inplace=True)  # Chaîne par défaut

# Forward fill / Backward fill
df['age'].fillna(method='ffill', inplace=True)  # Répéter la valeur précédente
df['age'].fillna(method='bfill', inplace=True)  # Utiliser la valeur suivante
```

Tri et agrégation

```
# Tri
df_sorted = df.sort_values('age')  # Tri croissant
df_sorted = df.sort_values('age', ascending=False)  # Décroissant
df_sorted = df.sort_values(['ville', 'age'])  # Multi-colonnes

# Agrégation
moyenne_age = df['age'].mean()
total = df['age'].sum()
min_max = df['age'].agg(['min', 'max', 'mean'])

# Groupby (groupe et agrège)
par_ville = df.groupby('ville')['age'].mean()
print(par_ville)

# Agrégations multiples
stats = df.groupby('ville').agg({
    'age': ['mean', 'min', 'max'],
    'nom': 'count'
})
print(stats)
```

Écriture CSV avec Pandas

```
# Écriture basique
df.to_csv('output.csv', index=False)

# Avec options
df.to_csv(
    'output.csv',
    sep=',',
    encoding='utf-8',
    index=False,           # Ne pas inclure l'index
    header=True,           # Inclure les en-têtes
    na_rep='NA',           # Représentation des NaN
    columns=['nom', 'age'] # Sélectionner des colonnes
)

# Ajouter à un fichier existant
df.to_csv('output.csv', mode='a', header=False, index=False)
```

Fichiers Excel - Introduction

Formats Excel

- **.xls** : Ancien format (Excel 97-2003)
- **.xlsx** : Format moderne (Excel 2007+)
- **.xlsm** : Avec macros

Différences avec CSV

- Plusieurs feuilles dans un fichier
- Formules Excel
- Formatage (couleurs, polices, bordures)
- Types de données plus riches
- Taille de fichier plus importante

Bibliothèques Python

- **OpenPyXL** : Lecture et écriture .xlsx (recommandé)
- **xlrd/xlwt** : Lecture/écriture .xls (legacy)
- **Pandas** : Interface haut niveau

OpenPyXL - Lecture basique

```python
from openpyxl import load_workbook

# Charger un fichier
wb = load_workbook('data.xlsx')

# Lister les feuilles
print(wb.sheetnames)  # ['Feuil1', 'Feuil2', ...]

# Sélectionner une feuille
ws = wb.active  # Feuille active
ws = wb['Feuil1']  # Par nom

# Lire une cellule
valeur = ws['A1'].value  # Coordonnées Excel
valeur = ws.cell(row=1, column=1).value  # Par numéros (commence à 1)

# Lire une plage
for row in ws['A1:C5']:
    for cell in row:
        print(cell.value, end='\t')
    print()

# Itérer sur toutes les lignes
for row in ws.iter_rows(min_row=1, values_only=True):
    print(row)
```

OpenPyXL - Écriture basique

```python
from openpyxl import Workbook

# Créer un nouveau classeur
wb = Workbook()
ws = wb.active
ws.title = "Ventes"

# Écrire dans des cellules
ws['A1'] = 'Produit'
ws['B1'] = 'Quantité'
ws['C1'] = 'Prix'

ws.cell(row=2, column=1, value='Laptop')
ws.cell(row=2, column=2, value=5)
ws.cell(row=2, column=3, value=899.99)

# Ajouter plusieurs lignes
data = [
    ['Souris', 10, 29.99],
    ['Clavier', 3, 79.99],
    ['Écran', 2, 299.99]
]
for row in data:
    ws.append(row)

# Sauvegarder
wb.save('output.xlsx')
```

OpenPyXL - Formatage

```python
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill, Alignment, Border, Side

wb = Workbook()
ws = wb.active

# Écrire données
ws['A1'] = 'Titre'

# Police
ws['A1'].font = Font(name='Arial', size=14, bold=True, color='FF0000')

# Couleur de fond
ws['A1'].fill = PatternFill(start_color='FFFF00', end_color='FFFF00', fill_type='solid')

# Alignement
ws['A1'].alignment = Alignment(horizontal='center', vertical='center')

# Bordures
thin_border = Border(
    left=Side(style='thin'),
    right=Side(style='thin'),
    top=Side(style='thin'),
    bottom=Side(style='thin')
)
ws['A1'].border = thin_border

# Largeur de colonne
ws.column_dimensions['A'].width = 20

wb.save('formatted.xlsx')
```

OpenPyXL - Formules

```python
from openpyxl import Workbook

wb = Workbook()
ws = wb.active

# En-têtes
ws['A1'] = 'Quantité'
ws['B1'] = 'Prix unitaire'
ws['C1'] = 'Total'

# Données
ws['A2'] = 5
ws['B2'] = 10.50

# Formule
ws['C2'] = '=A2*B2'

# Autres exemples de formules
ws['A10'] = '=SUM(A2:A9)'
ws['B10'] = '=AVERAGE(B2:B9)'
ws['C10'] = '=MAX(C2:C9)'

# Important : openpyxl n'évalue pas les formules
# Elles sont calculées à l'ouverture dans Excel

wb.save('with_formulas.xlsx')
```

Pandas avec Excel

Lecture Excel avec Pandas

```
import pandas as pd

# Lire la première feuille
df = pd.read_excel('data.xlsx')

# Lire une feuille spécifique
df = pd.read_excel('data.xlsx', sheet_name='Ventes')

# Lire plusieurs feuilles
all_sheets = pd.read_excel('data.xlsx', sheet_name=None)  # Dictionnaire
df1 = all_sheets['Feuil1']
df2 = all_sheets['Feuil2']

# Avec options
df = pd.read_excel(
    'data.xlsx',
    sheet_name='Ventes',
    header=0,              # Ligne d'en-tête
    skiprows=2,            # Ignorer les 2 premières lignes
    usecols='A:D',         # Colonnes à lire
    nrows=100              # Limiter le nombre de lignes
)
```

Pandas - Écriture Excel

```python
import pandas as pd

# DataFrame exemple
df = pd.DataFrame({
    'Produit': ['Laptop', 'Souris', 'Clavier'],
    'Quantité': [5, 10, 3],
    'Prix': [899.99, 29.99, 79.99]
})

# Écriture basique
df.to_excel('output.xlsx', index=False)

# Plusieurs feuilles
with pd.ExcelWriter('multi_sheets.xlsx') as writer:
    df.to_excel(writer, sheet_name='Ventes', index=False)
    df2.to_excel(writer, sheet_name='Stocks', index=False)

# Avec formatage (via ExcelWriter et engine)
with pd.ExcelWriter('formatted.xlsx', engine='openpyxl') as writer:
    df.to_excel(writer, sheet_name='Data', index=False)

    # Accéder à la feuille pour formater
    workbook = writer.book
    worksheet = writer.sheets['Data']
    worksheet.column_dimensions['A'].width = 20
```

Nettoyage de données

Problèmes courants dans les données

1. **Valeurs manquantes**

```
df.fillna(0)  # Remplacer par 0
df.dropna()   # Supprimer les lignes
```

2. **Doublons**

```
df.drop_duplicates()
df.drop_duplicates(subset=['email'])  # Basé sur une colonne
```

3. **Espaces inutiles**

```
df['nom'] = df['nom'].str.strip()  # Enlever espaces début/fin
```

4. **Casse (majuscules/minuscules)**

```
df['ville'] = df['ville'].str.upper()  # Tout en majuscules
df['email'] = df['email'].str.lower()  # Tout en minuscules
```

Transformation de données

```
# Conversion de types
df['age'] = df['age'].astype(int)
df['date'] = pd.to_datetime(df['date'])

# Extraction depuis dates
df['annee'] = df['date'].dt.year
df['mois'] = df['date'].dt.month
df['jour_semaine'] = df['date'].dt.dayofweek

# Manipulation de chaînes
df['nom_complet'] = df['prenom'] + ' ' + df['nom']
df['initiales'] = df['nom'].str[0] + df['prenom'].str[0]

# Découper une chaîne
df[['rue', 'ville']] = df['adresse'].str.split(',', expand=True)

# Apply - Appliquer une fonction personnalisée
def categoriser_age(age):
    if age < 18:
        return 'Mineur'
    elif age < 65:
        return 'Adulte'
    else:
        return 'Senior'

df['categorie'] = df['age'].apply(categoriser_age)
```

Jointures et fusion

```
# Deux DataFrames
clients = pd.DataFrame({
    'client_id': [1, 2, 3],
    'nom': ['Alice', 'Bob', 'Charlie']
})

commandes = pd.DataFrame({
    'commande_id': [101, 102, 103],
    'client_id': [1, 1, 2],
    'montant': [100, 150, 200]
})

# Jointure (merge)
result = pd.merge(commandes, clients, on='client_id', how='left')
print(result)

# Types de jointure
# - inner : Intersection (par défaut)
# - left : Toutes les lignes de gauche
# - right : Toutes les lignes de droite
# - outer : Union

# Concaténation
df_concat = pd.concat([df1, df2], ignore_index=True)  # Vertical
df_concat = pd.concat([df1, df2], axis=1)             # Horizontal
```

API REST

Qu'est-ce qu'une API ?

API = Application Programming Interface

Interface permettant à deux applications de communiquer.

Analogie : Un serveur dans un restaurant

- Vous (client) : Demandez un plat
- Serveur (API) : Transmet la demande à la cuisine
- Cuisine (serveur) : Prépare le plat
- Serveur (API) : Vous apporte le plat

API REST

- **REST** = Representational State Transfer
- Architecture web standardisée
- Utilise HTTP (comme les sites web)
- Échange de données en JSON (principalement)

Pourquoi utiliser des API ?

Cas d'usage

1. **Récupérer des données externes**
   - Météo, taux de change, actualités
   - Données publiques (gouvernement, open data)
2. **Intégration de services**
   - Paiement (Stripe, PayPal)
   - Messagerie (SendGrid, Twilio)
   - CRM (Salesforce, HubSpot)
3. **Automatisation**
   - Synchronisation entre systèmes
   - Workflows automatisés
4. **Microservices**
   - Communication entre services

HTTP - Rappels

Méthodes HTTP principales

- **GET** : Récupérer des données (lecture)
- **POST** : Créer une ressource
- **PUT** : Modifier complètement une ressource
- **PATCH** : Modifier partiellement une ressource
- **DELETE** : Supprimer une ressource

Codes de statut

- **2xx** : Succès (200 OK, 201 Created)
- **3xx** : Redirection
- **4xx** : Erreur client (400 Bad Request, 401 Unauthorized, 404 Not Found)
- **5xx** : Erreur serveur (500 Internal Server Error)

Structure d'une requête HTTP

```
GET /api/users/123 HTTP/1.1
Host: api.example.com
Authorization: Bearer token123
Content-Type: application/json

{
  "filter": "active"
}
```

Composants

- **Méthode** : GET, POST, etc.
- **URL** : Chemin de la ressource
- **Headers** : Métadonnées (authentification, type de contenu...)
- **Body** : Données envoyées (pour POST, PUT, PATCH)

Bibliothèque Requests

La bibliothèque HTTP de référence en Python

```
import requests
```

Installation

```
pip install requests
```

Avantages

- Simple et élégante
- Gestion automatique des sessions
- Support des méthodes HTTP
- Parsing JSON intégré
- Gestion des cookies et redirections

Première requête GET

```python
import requests

# Requête GET simple
response = requests.get('https://api.github.com')

# Afficher le code de statut
print(f"Statut : {response.status_code}")

# Afficher le contenu
print(response.text)  # Texte brut
print(response.json())  # Parsé en JSON (dict/list Python)

# Headers de la réponse
print(response.headers)

# Vérifier le succès
if response.status_code == 200:
    print("✓ Succès")
else:
    print("X Erreur")

# Ou utiliser
response.raise_for_status()  # Lève une exception si erreur
```

API publique - JSONPlaceholder

API de test gratuite

URL : `https://jsonplaceholder.typicode.com`

Endpoints disponibles

- `/posts` : Articles de blog
- `/users` : Utilisateurs
- `/comments` : Commentaires
- `/albums` : Albums
- `/photos` : Photos
- `/todos` : Tâches

Exemple

```python
import requests

# Récupérer tous les posts
response = requests.get('https://jsonplaceholder.typicode.com/posts')
posts = response.json()
print(f"Nombre de posts : {len(posts)}")
print(f"Premier post : {posts[0]}")
```

Paramètres de requête (Query Parameters)

Ajouter des paramètres à l'URL

```
import requests

# Méthode 1 : URL directement
response = requests.get('https://jsonplaceholder.typicode.com/posts?userId=1')

# Méthode 2 : Dictionnaire params (recommandé)
params = {
    'userId': 1,
    'id': 5
}
response = requests.get(
    'https://jsonplaceholder.typicode.com/posts',
    params=params
)

print(response.url)  # URL complète avec paramètres
posts = response.json()

# Filtrer plusieurs valeurs
params = {'userId': [1, 2]}  # userId=1&userId=2
```

JSON - Rappels

JSON = JavaScript Object Notation

Format d'échange de données léger et lisible.

Types de données

- **Object** : `{"key": "value"}` → dict Python
- **Array** : `[1, 2, 3]` → list Python
- **String** : `"texte"`
- **Number** : `42`, `3.14`
- **Boolean** : `true`, `false`
- **Null** : `null` → None Python

Exemple

```
{
  "nom": "Dupont",
  "age": 30,
  "actif": true,
  "adresse": {
    "ville": "Paris",
    "code_postal": "75001"
  },
  "hobbies": ["lecture", "sport"]
}
```

Manipulation JSON en Python

```python
import json

# JSON string vers Python dict
json_string = '{"nom": "Dupont", "age": 30}'
data = json.loads(json_string)
print(data['nom'])  # Dupont

# Python dict vers JSON string
data = {'nom': 'Martin', 'age': 25}
json_string = json.dumps(data)
print(json_string)  # {"nom": "Martin", "age": 25}

# Avec indentation (lisible)
json_string = json.dumps(data, indent=2)

# Lire depuis un fichier
with open('data.json', 'r') as f:
    data = json.load(f)

# Écrire dans un fichier
with open('output.json', 'w') as f:
    json.dump(data, f, indent=2)
```

Parsing JSON depuis API

```python
import requests

response = requests.get('https://api.github.com/users/github')

# Méthode 1 : response.json()
data = response.json()
print(data['login'])
print(data['public_repos'])

# Méthode 2 : json.loads (si besoin)
import json
data = json.loads(response.text)

# Navigation dans JSON imbriqué
response = requests.get('https://jsonplaceholder.typicode.com/users/1')
user = response.json()

print(user['name'])
print(user['address']['city'])  # JSON imbriqué
print(user['address']['geo']['lat'])  # Encore plus profond

# Liste d'objets
posts = requests.get('https://jsonplaceholder.typicode.com/posts').json()
for post in posts[:5]:
    print(f"{post['id']} - {post['title']}")
```

Requête POST

Créer une ressource

```python
import requests

url = 'https://jsonplaceholder.typicode.com/posts'

# Données à envoyer
nouveau_post = {
    'title': 'Mon titre',
    'body': 'Contenu de mon post',
    'userId': 1
}

# Requête POST
response = requests.post(url, json=nouveau_post)

print(f"Statut : {response.status_code}")  # 201 Created
print(f"Post créé : {response.json()}")

# Alternative : data avec json.dumps
import json
response = requests.post(
    url,
    data=json.dumps(nouveau_post),
    headers={'Content-Type': 'application/json'}
)
```

Requêtes PUT et PATCH

```python
import requests

base_url = 'https://jsonplaceholder.typicode.com/posts/1'

# PUT : Remplace complètement la ressource
updated_post = {
    'id': 1,
    'title': 'Titre mis à jour',
    'body': 'Nouveau contenu',
    'userId': 1
}
response = requests.put(base_url, json=updated_post)
print(response.json())

# PATCH : Modification partielle
partial_update = {
    'title': 'Juste le titre change'
}
response = requests.patch(base_url, json=partial_update)
print(response.json())

# DELETE : Supprimer
response = requests.delete(base_url)
print(f"Statut : {response.status_code}")  # 200
```

Headers HTTP

Headers personnalisés

```
import requests

# Headers communs
headers = {
    'User-Agent': 'MonApplication/1.0',
    'Accept': 'application/json',
    'Content-Type': 'application/json',
    'Accept-Language': 'fr-FR'
}

response = requests.get(
    'https://api.example.com/data',
    headers=headers
)

# Voir les headers de la réponse
print(response.headers)
print(response.headers['Content-Type'])

# Headers d'authentification
headers = {
    'Authorization': 'Bearer mon_token_secret'
}
response = requests.get(url, headers=headers)
```

Authentification - API Key

Méthode la plus simple

```python
import requests

# Méthode 1 : Dans les paramètres
params = {
    'apikey': 'votre_cle_api',
    'query': 'python'
}
response = requests.get('https://api.example.com/search', params=params)

# Méthode 2 : Dans les headers
headers = {
    'X-API-Key': 'votre_cle_api'
}
response = requests.get('https://api.example.com/data', headers=headers)

# Méthode 3 : Dans l'URL (déconseillé)
url = 'https://api.example.com/data?apikey=votre_cle_api'
response = requests.get(url)
```

⚠️ **Sécurité** : Ne jamais committer les clés API dans Git !

Gestion sécurisée des clés API

Utiliser des variables d'environnement

```
import os
from dotenv import load_dotenv

# Fichier .env
# API_KEY=votre_cle_secrete
# API_URL=https://api.example.com

# Charger les variables
load_dotenv()

# Utiliser
API_KEY = os.getenv('API_KEY')
API_URL = os.getenv('API_URL')

response = requests.get(
    f'{API_URL}/data',
    params={'apikey': API_KEY}
)
```

.gitignore

```
.env
config.py
secrets/
```

Authentification Bearer Token

Utilisé par OAuth 2.0, JWT

```python
import requests

# Token d'accès (obtenu après authentification)
access_token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'

# Header Authorization avec Bearer
headers = {
    'Authorization': f'Bearer {access_token}'
}

response = requests.get(
    'https://api.example.com/protected',
    headers=headers
)

if response.status_code == 401:
    print("Token invalide ou expiré")
elif response.status_code == 200:
    data = response.json()
    print(data)
```

API REST Countries - Exemple

API publique d'informations sur les pays

URL : `https://restcountries.com/v3.1`

```python
import requests

# Tous les pays
response = requests.get('https://restcountries.com/v3.1/all')
pays = response.json()
print(f"Nombre de pays : {len(pays)}")

# Un pays spécifique
response = requests.get('https://restcountries.com/v3.1/name/france')
france = response.json()[0]
print(f"Capitale : {france['capital'][0]}")
print(f"Population : {france['population']:,}")
print(f"Région : {france['region']}")

# Par code (plus rapide)
response = requests.get('https://restcountries.com/v3.1/alpha/fr')

# Filtrer les champs
response = requests.get('https://restcountries.com/v3.1/all?fields=name,capital,population')
```

Gestion des erreurs

```python
import requests
from requests.exceptions import RequestException, Timeout, ConnectionError

try:
    response = requests.get('https://api.example.com/data', timeout=5)
    response.raise_for_status()  # Lève HTTPError si code 4xx ou 5xx

    data = response.json()
    print(data)

except Timeout:
    print("⏱️ Timeout : L'API met trop de temps à répondre")

except ConnectionError:
    print("🔌 Erreur de connexion : Vérifiez votre réseau")

except requests.exceptions.HTTPError as e:
    print(f"❌ Erreur HTTP : {e}")
    print(f"Code : {response.status_code}")
    print(f"Message : {response.text}")

except RequestException as e:
    print(f"❌ Erreur générale : {e}")

except ValueError:
    print("⚠️ La réponse n'est pas du JSON valide")
```

Retry automatique

```python
import requests
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry

# Configuration retry
retry_strategy = Retry(
    total=3,                    # 3 tentatives
    backoff_factor=1,           # Attente : 1s, 2s, 4s
    status_forcelist=[429, 500, 502, 503, 504],  # Codes à retry
    allowed_methods=["GET", "POST"]
)

# Créer une session avec retry
adapter = HTTPAdapter(max_retries=retry_strategy)
session = requests.Session()
session.mount("http://", adapter)
session.mount("https://", adapter)

# Utiliser la session
try:
    response = session.get('https://api.example.com/data', timeout=5)
    data = response.json()
except requests.exceptions.RequestException as e:
    print(f"Échec après 3 tentatives : {e}")
```

Pagination

Récupérer des données paginées

```python
import requests

def get_all_pages(base_url, params=None):
    """Récupère toutes les pages d'une API paginée"""
    all_data = []
    page = 1

    while True:
        # Ajouter le numéro de page
        params_with_page = params.copy() if params else {}
        params_with_page['page'] = page

        response = requests.get(base_url, params=params_with_page)
        data = response.json()

        # Si plus de données, arrêter
        if not data or len(data) == 0:
            break

        all_data.extend(data)
        page += 1

        print(f"Page {page-1} récupérée : {len(data)} éléments")

    return all_data

# Utilisation
all_items = get_all_pages('https://api.example.com/items', {'per_page': 100})
```

OpenWeatherMap API

API météo populaire

1. Créer un compte : https://openweathermap.org/
2. Obtenir une API key gratuite

```python
import requests
import os

API_KEY = os.getenv('OPENWEATHER_API_KEY')
BASE_URL = 'https://api.openweathermap.org/data/2.5'

# Météo actuelle
ville = 'Paris'
url = f'{BASE_URL}/weather'
params = {
    'q': ville,
    'appid': API_KEY,
    'units': 'metric',  # Celsius
    'lang': 'fr'
}

response = requests.get(url, params=params)
data = response.json()

print(f"Météo à {ville} :")
print(f"Température : {data['main']['temp']}°C")
print(f"Ressenti : {data['main']['feels_like']}°C")
print(f"Description : {data['weather'][0]['description']}")
print(f"Humidité : {data['main']['humidity']}%")
```

Pipeline ETL

Architecture d'un pipeline ETL

Structure typique

```
projet_etl/
├── config/
│   ├── config.yaml
│   └── .env
├── src/
│   ├── extractors/
│   │   ├── csv_extractor.py
│   │   ├── api_extractor.py
│   │   └── excel_extractor.py
│   ├── transformers/
│   │   ├── cleaner.py
│   │   └── enricher.py
│   ├── loaders/
│   │   ├── csv_loader.py
│   │   └── excel_loader.py
│   └── utils/
│       ├── logger.py
│       └── validators.py
├── tests/
├── data/
│   ├── raw/
│   ├── processed/
│   └── output/
├── logs/
├── main.py
└── requirements.txt
```

Principes de conception

1. Modularité

- Séparer Extract, Transform, Load
- Fonctions réutilisables
- Composants indépendants

2. Configuration externe

- Paramètres dans fichiers de config
- Secrets dans variables d'environnement
- Pas de hardcoding

3. Gestion des erreurs

- Try/except appropriés
- Logging détaillé
- Récupération gracieuse

4. Idempotence

- Même résultat si exécuté plusieurs fois
- Pas d'effets de bord non contrôlés

Logging en Python

```python
import logging
from datetime import datetime

# Configuration du logger
def setup_logger(name, log_file=None, level=logging.INFO):
    """Configure un logger"""
    logger = logging.getLogger(name)
    logger.setLevel(level)

    # Format
    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S'
    )

    # Console handler
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    # File handler
    if log_file:
        file_handler = logging.FileHandler(log_file, encoding='utf-8')
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)

    return logger

# Utilisation
logger = setup_logger('ETL', 'logs/etl.log')
logger.info("Début du traitement")
logger.warning("Attention : données manquantes")
logger.error("Erreur lors de l'extraction")
```

Configuration avec YAML

Fichier config.yaml

```yaml
database:
  host: localhost
  port: 5432
  name: mydb

api:
  base_url: https://api.example.com
  timeout: 30
  retry: 3

etl:
  batch_size: 1000
  output_format: excel

paths:
  raw_data: data/raw
  processed_data: data/processed
  output: data/output
```

Lecture en Python

```python
import yaml

with open('config/config.yaml', 'r') as f:
    config = yaml.safe_load(f)

print(config['api']['base_url'])
print(config['etl']['batch_size'])
```

Classe Extractor

```python
import pandas as pd
import requests
from abc import ABC, abstractmethod

class BaseExtractor(ABC):
    """Classe de base pour les extracteurs"""

    def __init__(self, logger):
        self.logger = logger

    @abstractmethod
    def extract(self):
        """Méthode à implémenter par les classes filles"""
        pass

class CSVExtractor(BaseExtractor):
    """Extracteur pour fichiers CSV"""

    def extract(self, filepath):
        """Extrait données d'un CSV"""
        try:
            self.logger.info(f"Extraction de {filepath}")
            df = pd.read_csv(filepath)
            self.logger.info(f"✓ {len(df)} lignes extraites")
            return df
        except Exception as e:
            self.logger.error(f"❌ Erreur extraction CSV: {e}")
            raise

class APIExtractor(BaseExtractor):
    """Extracteur pour API REST"""

    def __init__(self, logger, base_url, api_key=None):
        super().__init__(logger)
        self.base_url = base_url
        self.api_key = api_key

    def extract(self, endpoint, params=None):
        """Extrait données d'une API"""
        try:
            self.logger.info(f"Extraction de {self.base_url}/{endpoint}")

            headers = {}
            if self.api_key:
                headers['Authorization'] = f'Bearer {self.api_key}'

            response = requests.get(
                f"{self.base_url}/{endpoint}",
                params=params,
                headers=headers,
                timeout=30
            )
            response.raise_for_status()

            data = response.json()
            self.logger.info(f"✓ Données extraites")
            return pd.DataFrame(data) if isinstance(data, list) else data

        except Exception as e:
            self.logger.error(f"❌ Erreur extraction API: {e}")
            raise
```

Classe Transformer

```python
import pandas as pd

class DataTransformer:
    """Transformateur de données"""

    def __init__(self, logger):
        self.logger = logger

    def clean(self, df):
        """Nettoie les données"""
        try:
            self.logger.info("Nettoyage des données")
            initial_count = len(df)

            # Supprimer doublons
            df = df.drop_duplicates()
            self.logger.info(f"  - {initial_count - len(df)} doublons supprimés")

            # Supprimer colonnes vides
            df = df.dropna(axis=1, how='all')

            # Trim espaces
            for col in df.select_dtypes(include=['object']).columns:
                df[col] = df[col].str.strip()

            self.logger.info(f"✓ Nettoyage terminé: {len(df)} lignes")
            return df

        except Exception as e:
            self.logger.error(f"❌ Erreur nettoyage: {e}")
            raise

    def validate(self, df, required_columns):
        """Valide la structure des données"""
        missing_cols = set(required_columns) - set(df.columns)
        if missing_cols:
            raise ValueError(f"Colonnes manquantes: {missing_cols}")

        self.logger.info("✓ Validation réussie")
        return True

    def enrich(self, df, enrichment_func):
        """Enrichit les données"""
        try:
            self.logger.info("Enrichissement des données")
            df = enrichment_func(df)
            self.logger.info("✓ Enrichissement terminé")
            return df
        except Exception as e:
            self.logger.error(f"❌ Erreur enrichissement: {e}")
            raise
```

Classe Loader

```python
import pandas as pd
from datetime import datetime

class DataLoader:
    """Chargeur de données"""

    def __init__(self, logger):
        self.logger = logger

    def load_csv(self, df, filepath, **kwargs):
        """Charge vers CSV"""
        try:
            self.logger.info(f"Chargement vers {filepath}")
            df.to_csv(filepath, index=False, **kwargs)
            self.logger.info(f"✓ {len(df)} lignes chargées")
        except Exception as e:
            self.logger.error(f"❌ Erreur chargement CSV: {e}")
            raise

    def load_excel(self, df, filepath, sheet_name='Data', **kwargs):
        """Charge vers Excel"""
        try:
            self.logger.info(f"Chargement vers {filepath}")

            with pd.ExcelWriter(filepath, engine='openpyxl') as writer:
                df.to_excel(writer, sheet_name=sheet_name, index=False, **kwargs)

            self.logger.info(f"✓ {len(df)} lignes chargées")
        except Exception as e:
            self.logger.error(f"❌ Erreur chargement Excel: {e}")
            raise

    def load_multiple_sheets(self, dataframes_dict, filepath):
        """Charge plusieurs feuilles Excel"""
        try:
            self.logger.info(f"Chargement multi-feuilles vers {filepath}")

            with pd.ExcelWriter(filepath, engine='openpyxl') as writer:
                for sheet_name, df in dataframes_dict.items():
                    df.to_excel(writer, sheet_name=sheet_name, index=False)
                    self.logger.info(f"  - Feuille '{sheet_name}': {len(df)} lignes")

            self.logger.info("✓ Chargement terminé")
        except Exception as e:
            self.logger.error(f"❌ Erreur chargement multi-feuilles: {e}")
            raise
```

Pipeline ETL - Orchestration

```python
class ETLPipeline:
    """Pipeline ETL complet"""

    def __init__(self, config, logger):
        self.config = config
        self.logger = logger
        self.extractor = None
        self.transformer = DataTransformer(logger)
        self.loader = DataLoader(logger)

    def run(self):
        """Exécute le pipeline"""
        try:
            self.logger.info("="*50)
            self.logger.info("DÉBUT DU PIPELINE ETL")
            self.logger.info("="*50)

            # Extract
            self.logger.info("\n[1/3] EXTRACTION")
            data = self._extract()

            # Transform
            self.logger.info("\n[2/3] TRANSFORMATION")
            data_transformed = self._transform(data)

            # Load
            self.logger.info("\n[3/3] CHARGEMENT")
            self._load(data_transformed)

            self.logger.info("\n" + "="*50)
            self.logger.info("✅ PIPELINE TERMINÉ AVEC SUCCÈS")
            self.logger.info("="*50)

        except Exception as e:
            self.logger.error(f"\n❌ PIPELINE ÉCHOUÉ: {e}")
            raise

    def _extract(self):
        """Logique d'extraction"""
        pass

    def _transform(self, data):
        """Logique de transformation"""
        pass

    def _load(self, data):
        """Logique de chargement"""
        pass
```

Tests unitaires

```python
import unittest
import pandas as pd
from src.transformers.cleaner import DataTransformer
from src.utils.logger import setup_logger

class TestDataTransformer(unittest.TestCase):
    """Tests pour DataTransformer"""

    def setUp(self):
        """Préparation avant chaque test"""
        self.logger = setup_logger('test')
        self.transformer = DataTransformer(self.logger)

        # Données de test
        self.df_test = pd.DataFrame({
            'nom': ['  Alice  ', 'Bob', 'Charlie', 'Bob'],  # Espaces, doublons
            'age': [25, 30, 35, 30],
            'ville': ['Paris', 'Lyon', 'Paris', 'Lyon']
        })

    def test_clean_removes_duplicates(self):
        """Teste la suppression des doublons"""
        df_cleaned = self.transformer.clean(self.df_test)
        self.assertEqual(len(df_cleaned), 3)  # 4 -> 3 lignes

    def test_clean_strips_spaces(self):
        """Teste le trim des espaces"""
        df_cleaned = self.transformer.clean(self.df_test)
        self.assertEqual(df_cleaned.iloc[0]['nom'], 'Alice')

    def test_validate_required_columns(self):
        """Teste la validation des colonnes"""
        required = ['nom', 'age']
        self.assertTrue(self.transformer.validate(self.df_test, required))

    def test_validate_missing_columns(self):
        """Teste la détection de colonnes manquantes"""
        required = ['nom', 'email']  # 'email' manquant
        with self.assertRaises(ValueError):
            self.transformer.validate(self.df_test, required)

if __name__ == '__main__':
    unittest.main()
```

Bonnes pratiques ETL

1. Idempotence

- Même résultat si exécuté plusieurs fois
- Utiliser des identifiants uniques
- Éviter les incréments sans contrôle

2. Gestion des erreurs

- Ne jamais laisser échouer silencieusement
- Logs détaillés
- Notifications en cas d'échec

3. Performance

- Traitement par batch si gros volumes
- Parallélisation si possible
- Indexation appropriée

4. Monitoring

- Métriques clés (durée, volume, erreurs)
- Alertes automatiques
- Dashboards de suivi

Optimisation et scaling

Pour de gros volumes

1. Traitement par chunks

```
chunk_size = 10000
for chunk in pd.read_csv('big_file.csv', chunksize=chunk_size):
    process(chunk)
```

2. Utiliser Dask (Pandas à grande échelle)

```
import dask.dataframe as dd
df = dd.read_csv('large_file.csv')
result = df.groupby('category')['amount'].sum().compute()
```

3. Base de données pour stockage intermédiaire

- SQLite pour petits volumes
- PostgreSQL, MySQL pour production

4. Orchestration avec Airflow

- Workflows complexes
- Dépendances entre tâches
- Retry automatique

Ressources et aller plus loin

Documentation

- Pandas : https://pandas.pydata.org/docs/
- Requests : https://requests.readthedocs.io/
- OpenPyXL : https://openpyxl.readthedocs.io/

APIs publiques pour pratiquer

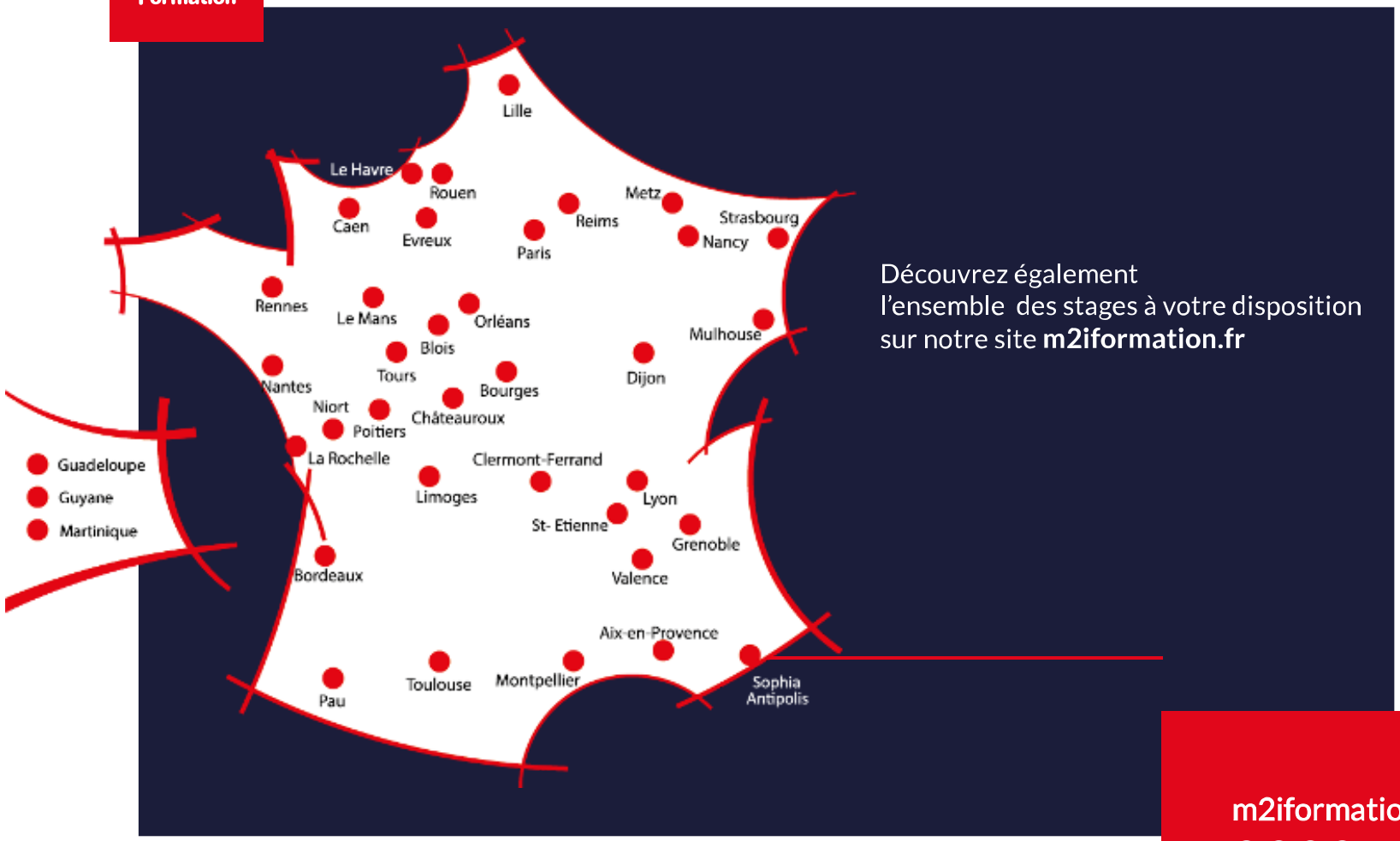
- JSONPlaceholder : https://jsonplaceholder.typicode.com/
- REST Countries : https://restcountries.com/
- OpenWeatherMap : https://openweathermap.org/api
- CoinGecko (crypto) : https://www.coingecko.com/en/api
- NASA : https://api.nasa.gov/

Outils professionnels

- Apache Airflow : Orchestration
- dbt : Transformation SQL
- Great Expectations : Validation de données
- Prefect : Orchestration moderne

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr