

Projet : Module 1

Architecture, Mémoire & Refactoring

M2 GER
Université de Nantes -- IGARUN

Introduction au Projet

Contexte

Nous avons appris à écrire des scripts linéaires (TD 1-4). Aujourd'hui, nous changeons de paradigme.

OBJECTIF ULTIME

Créer un outil capable de traiter n'importe quelle ville, sans toucher au code principal.

Cela demande de comprendre ce qui se passe **vraiment** dans la mémoire de l'ordinateur.

1. Théorie : La Variable

Qu'est-ce qu'une variable ?

Fondations

On dit souvent "C'est une boîte". C'est faux.

En Python, une variable est une **Étiquette** collée sur un objet en mémoire.

- `a = [1, 2]` : On crée une liste en mémoire (adresse 0x123). On colle l'étiquette 'a' dessus.
- `b = a` : On colle l'étiquette 'b' sur la **MÊME** adresse (0x123).



Référence vs Valeur

Démonstration par l'ID

Preuve

Python a une fonction `id()` qui révèle l'adresse mémoire.

```
ville_1 = "Nantes"  
ville_2 = ville_1  
  
print(id(ville_1)) # Ex: 1407234032  
print(id(ville_2)) # Ex: 1407234032 (Identique !)  
  
ville_2 = "Rennes"  
print(id(ville_2)) # L'adresse a changé !
```

Conclusion : Modifier une variable, c'est déplacer l'étiquette vers un nouvel objet.

Quiz Rapide : Mémoire

Test

Question 1

Si j'écris :

```
liste_a = [1, 2, 3]
liste_b = liste_a
liste_b.append(4)
```

Que contient liste_a ?

(Réfléchissez 10 secondes...)

Réponse : [1, 2, 3, 4]. Car a et b pointent vers le même objet en mémoire.

2. Le Fléau du "Hardcoding"

Code "En Dur" (Hardcoded)

Analyse

C'est l'ennemi de l'automatisation.

```
# Exemple typique de script "jetable"
gdf = gpd.read_file("data/Nantes.geojson") # ← En dur
buffer = gdf.buffer(50)                   # ← En dur
buffer.to_file("Resultat_Nantes.gpkg")    # ← En dur
```

Pour traiter "Rennes", vous devez modifier le code à 3 endroits. Sur un script de 200 lignes, c'est ingérable.

Le Principe de Configuration

Méthode

On sépare les **Données** (ce qui change) de la **Logique** (ce qui reste).

```
# --- CONFIGURATION (En haut du script) ---
VILLE = "Nantes"
DISTANCE_BUFFER = 50
FICHIER_IN = f"data/{VILLE}.geojson"

# --- LOGIQUE (Ne change jamais) ---
gdf = gpd.read_file(FICHIER_IN)
buffer = gdf.buffer(DISTANCE_BUFFER)
buffer.to_file(f"Resultat_{VILLE}.gpkg")
```

3. Mise en Pratique (Refactoring)

Exercice 1 : Analyse Critique

Pratique

CONSIGNE

Ouvrez votre Notebook du TD 3 (Analyse Nantes). Identifiez et listez toutes les valeurs "en dur" qui empêcheraient ce script de fonctionner pour Brest.

- Chemins de fichiers ?
- Noms de colonnes ?
- Codes EPSG ?
- Titres des graphiques ?

Correction Exercice 1

Solution

Les coupables habituels :

- "data/Nantes_Quartiers.geojson"
- 2154 (Peut-être que Brest utilise un autre système local ?)
- plt.title("Carte de Nantes")
- to_file("export_nantes.gpkg")

Nous allons les remplacer par des **Variables**.

Exercice 2 : Création de la Config

Pratique

Plutôt que des variables isolées, utilisons un **Dictionnaire**.

```
# À vous de jouer : Créez ce dictionnaire dans une nouvelle cellule
config = {
    "ville": "Nantes",
    "url_quartier": "https://data.nantes ... geojson",
    "url_parcs": "https://data.nantes ... geojson",
    "crs_cible": 2154,
    "buffer_dist": 50
}
```

Quiz : Accès au Dictionnaire

Test

Question 2

Comment accéder à la valeur 2154 dans le dictionnaire config défini précédemment ?

1. config[2154]
2. config.crs_cible
3. config["crs_cible"]

Réponse : La 3. Les dictionnaires s'interrogent par leur clé (entre guillemets).

4. Application : Chargement Dynamique

Le Script Dynamique

Code

Réécrivons le début du traitement.

```
import geopandas as gpd

print(f"Démarrage de l'analyse pour {config['ville']} ... ")

# Chargement dynamique
quartiers = gpd.read_file(config["url_quartier"])
parcs = gpd.read_file(config["url_parcs"])

# Projection dynamique
q_proj = quartiers.to_crs(epsg=config["crs_cible"])
```

Exercice 3 : L'Export Dynamique

Pratique

CONSIGNE

Écrivez la ligne de code qui exporte le résultat final (disons `gdf_final`) en GeoPackage.

Le nom du fichier doit être : `Analyse_Nantes_Resultat.gpkg`.

Mais attention : "Nantes" doit provenir de la variable `config["ville"]`.

Correction Exercice 3

Solution

```
# Utilisation des f-strings pour construire le nom de fichier
nom_fichier = f"Analyse_{config['ville']}_Resultat.gpkg"

# Export
gdf_final.to_file(nom_fichier, driver="GPKG")

print(f"Sauvegardé sous : {nom_fichier}")
```

Conclusion du Module 1

- Nous avons compris que les variables sont des références mémoire.
- Nous avons banni les valeurs "en dur".
- Nous avons centralisé tous les paramètres dans un dictionnaire config.

Mais... nous avons toujours un script qui ne traite qu'une seule ville à la fois.

Pour traiter Nantes, Rennes et Brest en même temps, il va falloir créer une **Liste de Dictionnaires**.

C'est le sujet du **Module 2**.

Projet : Module 2

Structures de Données (Listes & Dictionnaires)

M2 GER
Université de Nantes -- IGARUN

Objectifs du Module 2

Fondations

Pour gérer plusieurs villes, nous devons organiser nos données. Des variables simples (`ville1`, `ville2`) ne suffisent plus.

PROGRAMME

- **Théorie** : Différence fondamentale entre Liste (Séquence) et Dictionnaire (Mapping).
- **Concept** : Le format JSON et son lien avec Python.
- **Pratique** : Construction de la structure de données pour l'automatisation.

1. Théorie : Les Listes

La Liste : Une Séquence Ordonnée

Théorie

Une liste stocke des éléments dans un ordre précis.

On accède aux données par leur **position (Index)**.

- Le premier est 0.
- Le dernier est -1.

```
# Une liste de villes
villes = ["Nantes", "Rennes",
          "Brest"]

print(villes[0]) # "Nantes"
print(villes[2]) # "Brest"
```

Problème : Si je veux la population de Nantes, je ne sais pas où elle est rangée.

2. Théorie : Les Dictionnaires

Le Dictionnaire : Clé / Valeur

Théorie

Un dictionnaire (ou Hash Map) stocke des données associées à une **Clé unique**.

L'ordre n'a pas d'importance. C'est le nom de l'étiquette qui compte.

C'est la structure idéale pour des **attributs**.

```
# Un objet "Ville"
nantes = {
    "nom": "Nantes",
    "pop": 320000,
    "coords": [-1.55, 47.21]
}

print(nantes["pop"]) # 320000
```

POURQUOI LES DICTIONNAIRES SONT RAPIDES ?

Imaginez chercher un mot dans un livre (Liste) page par page. C'est long.

Imaginez le chercher dans un dictionnaire alphabétique. C'est instantané.

En Python, trouver une valeur par sa clé est une opération quasi-instantanée ($O(1)$), quelle que soit la taille des données.

3. Structures Imbriquées (JSON)

Liste de Dictionnaires

Architecture

Pour notre projet, nous avons besoin d'une liste de villes, où chaque ville est un dictionnaire de propriétés.

C'est exactement la structure du format **GeoJSON** que vous utilisez en Webmapping.

```
projet = [
    {"nom": "Nantes", "code": 44},
    {"nom": "Rennes", "code": 35}
]
```

Accéder aux Données Imbriquées

Pratique

Comment récupérer le code de Rennes ?

```
# 1. On accède au 2ème élément de la liste (Index 1)
ville_b = projet[1]

# 2. On accède à la clé "code"
code_b = ville_b["code"]

# En une ligne :
print(projet[1]["code"])
```

4. Mise en Pratique

Exercice 1 : Configuration Complète

Pratique

CONSIGNE

Créez une structure de données `config` qui contient une LISTE de 3 DICTIONNAIRES (Nantes, Rennes, Brest).

Pour chaque ville, définissez :

- "nom" (String)
- "fichier_quartier" (String - lien Open Data)
- "fichier_parcs" (String - lien Open Data)
- "epsg_local" (Int - 2154)

Correction Exercice 1

Solution

```
config = [
    {
        "nom": "Nantes",
        "fichier_quartier": "https://data.nantes ... geojson",
        "fichier_parcs": "https://data.nantes ... geojson",
        "epsg_local": 2154
    },
    {
        "nom": "Rennes",
        "fichier_quartier": "https://data.rennes ... geojson",
        "fichier_parcs": "https://data.rennes ... geojson",
        "epsg_local": 2154
    }
]
```

Exercice 2 : Itération Manuelle

Pratique

Sans utiliser de boucle (pour l'instant), écrivez un script qui charge les données de la première ville de la liste, puis de la deuxième.

```
# Ville 1
v1 = config[0]
gdf1 = gpd.read_file(v1["fichier_quartier"])

# Ville 2
v2 = config[1]
gdf2 = gpd.read_file(v2["fichier_quartier"])
```

Cela vous force à comprendre la structure avant d'automatiser.

5. Conclusion & Suite

Ce que vous avez appris

Bilan

- **Liste** : Collection ordonnée [].
- **Dictionnaire** : Collection nommée { }.
- **Imbrication** : La base des formats de données modernes (JSON).

PROCHAIN MODULE

Maintenant que nos données sont structurées, comment créer un "moteur" qui les traite ? Nous allons construire la **Fonction d'Analyse** (Module 3).