

# Projet : Module 3

Fonctions & Modularité

**M2 GER**  
Université de Nantes -- IGARUN

# Objectifs du Module 3

Fondations

Notre code fonctionne pour Nantes (Module 1) et nous avons structuré nos données (Module 2). Mais copier-coller le code pour Rennes et Brest est interdit.

## PROGRAMME

- **Théorie** : Le concept de la "Boîte Noire" (Input -> Magic -> Output).
- **Concept** : La portée des variables (Scope) : pourquoi ma variable n'existe pas ?
- **Pratique** : Transformation du script "Nantes" en une fonction générique `analyser_ville()`.

# 1. Théorie : La Fonction

---

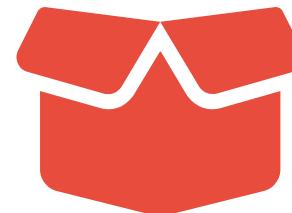
# L'Anatomie d'une Fonction

Théorie

---

Une fonction est un bloc de code nommé et réutilisable.

- **Input (Arguments)** : Ce qu'on lui donne (ex: Farine, Oeufs).
- **Corps** : Ce qu'elle fait (Mélanger, Cuire).
- **Output (Return)** : Ce qu'elle rend (Gâteau).



Input  $\rightarrow$  [ Boîte Noire ]  
 $\rightarrow$  Output

# La notion de Scope (Portée)

Concept Clé

C'est l'erreur n°1 des débutants.

```
def ma_fonction():
    x = 10  # Variable LOCALE : Elle naît et meurt ici

ma_fonction()
print(x)  # ERREUR ! x n'existe pas à l'extérieur.
```

Une variable créée DANS une fonction est invisible pour le reste du programme. Si vous voulez récupérer sa valeur, il faut utiliser return.

# Return vs Print

Nuance

## PRINT

Affiche du texte à l'écran pour l'humain.  
L'ordinateur ne peut pas récupérer la valeur pour un calcul suivant.  
C'est une impasse.

## RETURN

Renvoie la valeur au programme principal.  
C'est indispensable pour chaîner les traitements (ex: le résultat de la fonction devient l'entrée d'une autre).

## 2. Construction de la Fonction

---

# Identifier les Arguments

Analyse

Regardez votre script du Module 1. Quelles variables changent à chaque ville ?

- Le Nom de la ville.
- L'URL des Quartiers.
- L'URL des Parcs.

Ce seront les **arguments** (les entrées) de notre fonction.

```
def analyser_ville(nom_ville, url_q, url_p):  
    # Le code de traitement ira ici  
    pass
```

# Exercice 1 : Encapsulation

Pratique

## CONSIGNE

Prenez tout le code de traitement (chargement, projection) que vous avez validé au Module 1.

Mettez-le à l'intérieur d'une fonction `analyser_ville`.

Remplacez les variables fixes par les arguments de la fonction.

# Correction Exercice 1 (Squelette)

Solution

```
def analyser_ville(nom, url_q, url_p):
    print(f"--- Analyse de {nom} ---")

    # 1. Chargement (On utilise les arguments url_q et url_p)
    quartiers = gpd.read_file(url_q)
    parcs = gpd.read_file(url_p)

    # 2. Projection (Lambert-93 toujours)
    q_proj = quartiers.to_crs(epsg=2154)
    p_proj = parcs.to_crs(epsg=2154)

    return q_proj, p_proj
```

## Exercice 2 : Test Unitaire

Pratique

Une fonction ne sert à rien si on ne l'appelle pas.

### CONSIGNE

Utilisez votre dictionnaire de configuration (créé au Module 2, index 0 pour Nantes) pour tester votre fonction.

```
# Récupérer la config de Nantes
conf = config[0]

# Appel de la fonction
q_resultat, p_resultat = analyser_ville(
    conf["nom"],
    conf["fichier_quartier"],
    conf["fichier_parcs"]
)
```

### **3. Amélioration & Robustesse**

---

# Arguments par défaut

Astuce

On peut définir des valeurs par défaut pour simplifier l'utilisation. Par exemple, si on travaille toujours en France, le code EPSG sera souvent 2154.

```
# epsg=2154 est la valeur par défaut si on ne précise rien
def analyser_ville(nom, url_q, url_p, epsg=2154):

    quartiers = quartiers.to_crs(epsg=epsg)
    # ...
```

Un bon développeur documente son code pour expliquer ce que fait la fonction.

```
def analyser_ville(nom, url_q, url_p):
    """
    Charge, projette et prépare les données d'une ville.

    Args:
        nom (str): Nom de la ville.
        url_q (str): URL du GeoJSON quartiers.
        url_p (str): URL du GeoJSON parcs.

    Returns:
        GeoDataFrame: Les quartiers et parcs projetés.
    """
    # ... code ...
```

# Exercice 3 : La Fonction Complète

Pratique

Intégrez tout le traitement spatial (Buffer, Export) dans la fonction.

## CONSIGNE

La fonction doit : 1. Calculer les surfaces des parcs. 2. Exporter le fichier GPKG avec un nom dynamique (`f"Resultat_{nom}.gpkg"`). 3. Retourner le GeoDataFrame des parcs modifiés.

## 4. Conclusion & Suite

---

# Ce que vous avez construit

Bilan

Vous avez créé une "boîte noire" analyser\_ville.

- Elle est **modulaire** : elle ne dépend pas de variables globales externes.
- Elle est **réutilisable** : elle marche pour n'importe quelle ville (Nantes, Rennes, Brest).
- Elle est **propre** : elle sépare la logique des données.

## PROCHAIN MODULE

Comment utiliser cette fonction non pas une fois, mais 3, 10 ou 100 fois automatiquement ? Nous verrons les **Boucles** et la **Géométrie Avancée**.

# Projet : Module 4

Géométrie & Algorithmes Spatiaux

**M2 GER**  
Université de Nantes -- IGARUN

# Objectifs du Module 4

Fondations

Notre fonction commence à ressembler à quelque chose. Maintenant, il faut mettre le "moteur" spatial dedans.

## PROGRAMME

- **Théorie** : Ce qui se passe "sous le capot" de GeoPandas (Shapely, GEOS).
- **Mathématiques** : Pourquoi on ne peut pas calculer une aire sur des degrés (WGS84).
- **Pratique** : Intégrer la logique Buffer + Intersection dans notre fonction générique.

# 1. Théorie : Le Moteur Géométrique

---

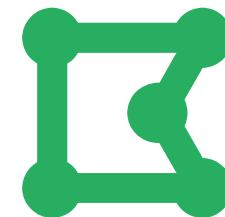
# Shapely & GEOS

Théorie

---

GeoPandas ne fait pas les calculs lui-même. Il délègue à **Shapely**, qui délègue à **GEOS** (C++).

- Un **Point** est un tuple (x, y).
- Un **Polygone** est une liste de points fermée.



**Algorithmes Vectoriels**

# L'Importance du CRS

Rappel Critique

En informatique,  $\text{distance}((0,0), (1,1)) = \sqrt{2} \approx 1.414$ .

Si vos coordonnées sont en degrés (lat/lon) :

- X et Y n'ont pas la même échelle.
- Le résultat "1.414 degrés" ne veut rien dire physiquement.

## IMPÉRATIF

Toujours projeter en système métrique (Lambert-93) AVANT tout calcul géométrique (Buffer, Area).

## **2. Algorithmes : Buffer & Intersection**

---

# L'Algorithme du Buffer

Mécanique

Comment l'ordinateur crée un cercle autour d'un point ?

1. Il prend le centre ( $x, y$ ).
2. Il calcule N points à une distance R.
3. Il relie ces points pour former un polygone.

```
# Plus la résolution est élevée, plus le cercle est "rond"  
gdf.geometry.buffer(50, resolution=16)
```

# L'Algorithme d'Intersection (SJoin)

Mécanique

---

Pour savoir si un Point est dans un Polygone (Point-in-Polygon), l'ordinateur lance un rayon imaginaire.

- S'il croise les bords du polygone un nombre **impair** de fois  $\rightarrow$  Dedans.
- S'il croise un nombre **pair** de fois  $\rightarrow$  Dehors.

C'est coûteux en calcul ! D'où l'intérêt des index spatiaux (R-Tree) gérés automatiquement par GeoPandas.

### **3. Intégration dans la Fonction**

---

# Mise à jour de notre fonction

Pratique

Reprendons notre fonction analyser\_ville du Module 3. Nous allons y ajouter la "vraie" intelligence spatiale.

```
def analyser_ville(nom, url_q, url_p):
    # ... (Chargement & Projection déjà faits) ...

    # 1. Création des Surfaces (Buffers)
    p_buffer = p_proj.copy()
    p_buffer["geometry"] = p_proj.buffer(50)

    # 2. Calcul des Aires
    p_buffer["surf_m2"] = p_buffer.area

    # ... (Suite au prochain slide)
```

# Exercice 1 : Spatial Join Intégré

Pratique

## CONSIGNE

Complétez la fonction pour : 1. Effectuer la jointure spatiale entre les buffers et les quartiers. 2. Agréger les surfaces par quartier (groupby). 3. Retourner le tableau final.

*Rappel : gpd.sjoin(left, right, op="intersects")*

# Correction Exercice 1

Solution

```
# Suite de la fonction...

# 3. Jointure Spatiale
join = gpd.sjoin(p_buffer, q_proj, predicate="intersects")

# 4. Agrégation
# On groupe par l'index des quartiers (pour être sûr)
stats = join.groupby("index_right")["surf_m2"].sum()

# 5. Fusion pour récupérer la géométrie des quartiers
final = q_proj.join(stats, rsuffix="_parcs").fillna(0)

return final
```

## Exercice 2 : Test Complet

Pratique

Testez votre nouvelle fonction "intelligente" sur Nantes.

```
# Test
gdf_nantes = analyser_ville(
    "Nantes",
    config[0]["url_quartier"],
    config[0]["url_parcs"]
)

# Vérification
print(gdf_nantes[["nom", "surf_m2"]].head())
```

Si vous voyez des surfaces non nulles, votre moteur spatial fonctionne !

## 4. Conclusion & Suite

---

# Ce que vous avez construit

Bilan

Votre fonction est maintenant un véritable outil SIG.

- Elle gère la physique (Projection).
- Elle gère la géométrie (Buffer).
- Elle gère la topologie (Intersection).

## PROCHAIN MODULE

Maintenant que le moteur tourne, il faut l'alimenter en carburant (les données des autres villes) de manière automatique. Nous verrons les **Boucles**.