[Ignorer et passer au contenu](https://chatgpt.com/g/g-p-68d400ac65088191bb252886ca964a3b/c/69004d5a-71a0-832d-879e-cdfd5d5f4090" \l "main)

**Historique de chat**

**Vous avez dit :**

python creation d'un singleton

**ChatGPT a dit :**

Un **singleton** en Python est une classe dont **une seule instance** peut exister.  
Il y a plusieurs manières de le faire — certaines plus “pythoniques” que d’autres. Voici les trois plus courantes :

**🧩 1. Avec une variable de classe (méthode simple et suffisante)**

class Singleton:

\_instance = None

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

**Usage :**

a = Singleton()

b = Singleton()

print(a is b) # True

👉 Avantage : clair, fonctionne bien.  
👉 Inconvénient : pas thread-safe (dans un contexte multithread, deux instances peuvent se créer).

**🔒 2. Avec un décorateur**

def singleton(cls):

instances = {}

def getinstance(\*args, \*\*kwargs):

if cls not in instances:

instances[cls] = cls(\*args, \*\*kwargs)

return instances[cls]

return getinstance

@singleton

class MaClasse:

pass

**Usage :**

x = MaClasse()

y = MaClasse()

print(x is y) # True

👉 Avantage : élégant et réutilisable.  
👉 Inconvénient : masque la nature “classe” de l’objet (par ex. isinstance(x, MaClasse) ne marche plus sans ajustement).

**🧠 3. Avec un module Python**

En pratique, **le plus simple singleton en Python est un module** :

# fichier config.py

param = "valeur"

Puis :

import config

print(config.param)

Chaque import pointe sur **la même instance** du module — Python gère cela nativement.  
👉 C’est la solution la plus propre dans beaucoup de cas.

Tu veux que je te montre une version thread-safe aussi (utile si le code tourne en parallèle) ?

**Vous avez dit :**

l'utilisation des singletons n'est plus à la mode.Est ce exact ?

**ChatGPT a dit :**

Oui, plutôt juste — ou disons que leur **emploi est devenu suspect** dans beaucoup de contextes.

Les singletons étaient populaires à une époque où on voulait garantir qu’un objet comme une configuration, une connexion à la base, un logger… n’ait qu’une seule instance. En pratique, ça a souvent amené :

* des **dépendances cachées** (le code dépend d’un état global sans le dire),
* des **tests unitaires difficiles** (les singletons gardent leur état entre tests),
* et une **couplage fort** (tout le monde parle à la même instance).

Aujourd’hui, on préfère d’autres approches :

* **injection de dépendances** : tu passes l’objet dont tu as besoin aux fonctions/classes qui l’utilisent ;
* **modules ou contextes** : Python fait déjà du singleton implicite au niveau des modules ;
* ou **usages conscients de globales**, mais en les isolant clairement.

Les singletons ne sont donc pas “mauvais” par essence, juste souvent **le signe d’un design trop rigide**.

Tu penses à un cas précis où tu envisages d’en utiliser un ?

**Vous avez dit :**

Oui. Dans mon cas, je cree un ecran unique pour toute l'application à partir duquel je lance le menu qui doit me donner acces à l'ensemble des fonctionnalités de l'application y compris la fermeture de l'écran et l'arret de l'application

**ChatGPT a dit :**

Là, ton besoin est clair : tu veux un **point d’ancrage unique pour l’interface**, pas un singleton abstrait.

Dans ce cas, ce que tu décris — une seule fenêtre principale (root) qui reste vivante pendant tout le cycle de l’application — est déjà le fonctionnement naturel de Tkinter.  
Tu n’as **pas besoin** de forcer le concept de singleton :

import tkinter as tk

root = tk.Tk()

root.title("Mon application")

# le menu, les frames, les écrans… tout part de là

# ...

root.mainloop()

Ensuite, tu passes root en paramètre à tes autres classes ou fonctions qui doivent manipuler l’écran principal.  
C’est plus simple, plus lisible, et tout aussi “unique”.

Tu peux garder un petit gestionnaire global si tu veux centraliser les accès — par exemple :

class AppContext:

def \_\_init\_\_(self, root):

self.root = root

self.menu = None

self.current\_frame = None

et tu passes une instance de AppContext à ce qui en a besoin.  
Ça t’évite les pièges du vrai singleton, tout en gardant ton “écran unique”.

Tu veux que je te montre comment articuler ce AppContext avec un menu et des écrans Tkinter ?

**Vous avez dit :**

dans ce cas, si j'importe le module qui contient le script que tu décris, je crée un deuxième conteneur

**ChatGPT a dit :**

Exact — c’est là que Tkinter peut piéger :  
chaque tk.Tk() crée **un nouveau conteneur racine**, donc une **deuxième fenêtre** indépendante.

La bonne pratique :

* Tu ne fais appel à tk.Tk() **qu’une seule fois**, dans ton point d’entrée principal.
* Dans les autres modules, tu utilises tk.Toplevel() si tu veux de nouvelles fenêtres,  
  ou bien tu récupères la référence à root déjà existante.

Un schéma typique :

**main.py**

import tkinter as tk

from ui import AppUI

def main():

root = tk.Tk()

app = AppUI(root)

root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**ui.py**

import tkinter as tk

from tkinter import ttk

class AppUI:

def \_\_init\_\_(self, root):

self.root = root

self.\_build\_ui()

def \_build\_ui(self):

frame = ttk.Frame(self.root)

frame.pack()

ttk.Label(frame, text="Interface principale").pack()

Quand tu veux accéder à root ailleurs, tu le fais passer, pas l’importer.  
Par exemple :

def ouvrir\_fenetre(context):

top = tk.Toplevel(context.root)

tk.Label(top, text="Nouvelle fenêtre").pack()

Ça garde une seule racine, pas de singleton caché, et tout reste propre.

Tu veux que je te montre comment brancher un menu dans cette structure ?

**Vous avez dit :**

ok, l'astuce consiste à creer le conteneur dans main qu'en principe on n'importe pas

**ChatGPT a dit :**

Exactement.  
Tu mets la création du conteneur (le tk.Tk()) **dans un fichier qu’on exécute, pas qu’on importe** — typiquement ton main.py.

Ça évite deux écueils :

* l’importation involontaire qui recrée une fenêtre ;
* le lancement automatique de mainloop() alors que tu voulais juste accéder à une classe.

Tout ce qui construit l’interface (frames, menus, etc.) vit dans des modules séparés, mais **ils reçoivent root** depuis main, ils ne le créent pas.

C’est une forme de “singleton naturel” : il n’existe qu’un seul root, mais parce que **tu le décides par structure**, pas par un pattern.

Tu veux que je t’aide à organiser les fichiers du projet autour de cette idée ?

**Vous avez dit :**

oui

**ChatGPT a dit :**

Voici une structure claire et modulaire — assez typique pour une appli Tkinter sérieuse :

mon\_app/

│

├── main.py

├── ui/

│ ├── \_\_init\_\_.py

│ ├── app\_ui.py

│ ├── menu.py

│ └── pages/

│ ├── accueil.py

│ ├── parametres.py

│ └── ...

└── core/

├── \_\_init\_\_.py

├── context.py

├── data.py

└── ...

**1. main.py — point d’entrée**

import tkinter as tk

from ui.app\_ui import AppUI

from core.context import AppContext

def main():

root = tk.Tk()

root.title("Mon application")

context = AppContext(root)

AppUI(context) # construit l'interface et le menu

root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**2. core/context.py — partage des objets communs**

class AppContext:

def \_\_init\_\_(self, root):

self.root = root

self.menu = None

self.current\_page = None

# éventuellement d'autres éléments partagés (config, db, etc.)

**3. ui/app\_ui.py — configuration générale de l’interface**

from tkinter import ttk

from ui.menu import build\_menu

from ui.pages.accueil import AccueilPage

class AppUI:

def \_\_init\_\_(self, context):

self.context = context

self.root = context.root

self.\_setup\_ui()

def \_setup\_ui(self):

self.context.menu = build\_menu(self.root, self.context)

frame = ttk.Frame(self.root)

frame.pack(fill="both", expand=True)

self.context.current\_page = AccueilPage(frame, self.context)

**4. ui/menu.py**

import tkinter as tk

def build\_menu(root, context):

menubar = tk.Menu(root)

fichier\_menu = tk.Menu(menubar, tearoff=0)

fichier\_menu.add\_command(label="Quitter", command=root.quit)

menubar.add\_cascade(label="Fichier", menu=fichier\_menu)

root.config(menu=menubar)

return menubar

**5. ui/pages/accueil.py**

from tkinter import ttk

class AccueilPage:

def \_\_init\_\_(self, parent, context):

label = ttk.Label(parent, text="Bienvenue dans l'application")

label.pack(padx=20, pady=20)

Avec ça :

* main.py **crée le seul et unique root**.
* Les modules de ui **ne font qu’ajouter des éléments** à cet écran.
* Le context sert de fil conducteur entre les morceaux (utile si tu veux, plus tard, gérer la base ou un état global).

Souple, testable, pas de singleton.

Tu veux que je te montre comment ajouter une page secondaire (par exemple “Paramètres”) accessible depuis le menu ?

**Vous avez dit :**

quel est le role de context.py

**ChatGPT a dit :**

context.py sert de **passeur silencieux** entre les différentes parties de ton appli.

Dans une appli Tkinter, tu as souvent besoin que plusieurs modules se partagent des objets communs :

* la fenêtre principale (root),
* le menu,
* la page courante,
* parfois une connexion à la base, une config, un cache, etc.

Plutôt que d’utiliser des variables globales ou des singletons pour ça, tu regroupes ces éléments dans un **contexte** :

class AppContext:

def \_\_init\_\_(self, root):

self.root = root

self.menu = None

self.current\_page = None

self.db = None

self.config = None

Ensuite, tu passes ce context aux classes et fonctions qui en ont besoin.  
Elles peuvent lire ou modifier les éléments partagés sans recréer quoi que ce soit.

Exemple concret :

# dans main.py

context = AppContext(root)

AppUI(context)

# dans menu.py

def build\_menu(root, context):

fichier\_menu.add\_command(

label="Accueil",

command=lambda: context.current\_page.show\_home()

)

Le contexte joue donc un rôle de **point central explicite**, mais sans rigidité :  
tu gardes une seule source de vérité (le context),  
et chaque module reste découplé du reste — il n’a pas besoin d’importer directement root ou db.

En somme : c’est ton **état global rendu propre**.

Tu veux que je t’illustre ça avec un exemple où le menu change la “page” affichée via le context ?

**Vous avez dit :**

la structure de mon projet est la suivante, qu'en penses-tu ? mon\_projet | |- mon\_envrt |- Resutlats-| |- mes\_resultats.pdf | |- sources --| |- ma\_source.xlsx | |-src-| | |\_\_init\_\_.py | | utils | |-\_\_init\_\_.py | |- tous\_mes\_modules.py | |- menu.py |-main.py |-variables\_communes.py

**ChatGPT a dit :**

Claire, propre et fonctionnelle.  
Tu restes dans une logique simple, sans trop morceler — ce qui est souvent mieux au début.

Quelques remarques légères, juste pour la clarté future :

* **src/utils** : si menu.py contient des éléments Tkinter, le mot “utils” prête un peu à confusion — on s’attend à y trouver des fonctions génériques, pas de l’interface. Tu pourrais un jour séparer ça en src/ui/menu.py si le projet grandit.
* **variables\_communes.py** : ce fichier joue visiblement le rôle de ton context. Rien à redire tant qu’il ne devient pas une “poubelle globale”. Si tu sens qu’il s’alourdit (ex. des configs, de l’état de session, des constantes…), tu pourras plus tard le scinder en :
  + config.py (valeurs fixes),
  + context.py (objets partagés),
  + constantes.py (textes, chemins, etc.).
* Le dossier **Resultats** avec un “R” majuscule : ça marche, mais sur certains OS, la casse peut semer la confusion. “resultats” tout en minuscules reste plus conventionnel en Python.

Sinon, la hiérarchie est lisible : tu sépares données, code et sorties — parfait pour du long terme.

Tu veux que je t’aide à harmoniser les imports entre main.py, variables\_communes.py et les modules dans src/utils ?

**Vous avez dit :**

comment l'adapter à la structure tkinter que tu m'as proposée

**ChatGPT a dit :**

Tu peux garder ta base, il suffit juste d’**ordonner un peu les rôles** pour coller à la logique Tkinter “un seul root / modules d’interface bien rangés”.  
Voici une version adaptée de ta structure :

mon\_projet/

│

├── main.py

├── variables\_communes.py # → devient ton AppContext (ou config)

│

├── src/

│ ├── \_\_init\_\_.py

│ ├── ui/

│ │ ├── \_\_init\_\_.py

│ │ ├── app\_ui.py # structure générale de l'écran

│ │ ├── menu.py # construction du menu

│ │ └── pages/

│ │ ├── accueil.py

│ │ ├── parametres.py

│ │ └── ...

│ │

│ ├── utils/

│ │ ├── \_\_init\_\_.py

│ │ └── tous\_mes\_modules.py # fonctions générales, indépendantes de l’UI

│ │

│ └── core/

│ ├── \_\_init\_\_.py

│ ├── context.py # ton “variables\_communes” allégé

│ └── data.py # ex. gestion de tes sources Excel / SQLite

│

├── sources/

│ └── ma\_source.xlsx

│

└── resultats/

└── mes\_resultats.pdf

**Comment ça s’articule**

**main.py**

* C’est le *point d’entrée unique*.
* Crée root, instancie ton AppContext, lance l’UI principale (AppUI).

**src/core/context.py**

* Centralise root, menu, page courante, chemins, etc.
* Remplace avantageusement variables\_communes.py (qui devient une coquille plus fine ou disparaît).

**src/ui/**

* Contient toute la partie Tkinter.
* app\_ui.py assemble la fenêtre principale.
* menu.py gère le menu.
* pages/ contient les différents écrans.

**src/utils/**

* Regroupe tes fonctions métiers, calculs, manipulations de données — rien d’interface ici.

**sources/ et resultats/**

* Tu les laisses tels quels : c’est ton entrée/sortie métier.

Cette organisation garde ton empreinte mais introduit un **cloisonnement clair entre l’interface, le cœur et les utilitaires**.  
Tu peux même, plus tard, lancer des tests unitaires sur utils sans jamais ouvrir une fenêtre Tkinter.

Tu veux que je te montre comment réécrire ton main.py et ton variables\_communes.py pour qu’ils s’intègrent dans cette arborescence ?

**Vous avez dit :**

oui

**ChatGPT a dit :**

Parfait. Voici une version légère et cohérente avec la structure ci-dessus.

**main.py**

import tkinter as tk

from src.core.context import AppContext

from src.ui.app\_ui import AppUI

def main():

# Fenêtre principale

root = tk.Tk()

root.title("Mon application de copropriété")

# Contexte partagé

context = AppContext(root)

# Interface principale

AppUI(context)

# Boucle Tkinter

root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Idées clés :**

* tk.Tk() ne vit qu’ici.
* Rien d’autre ne crée de fenêtre racine.
* Les modules n’ont besoin que du context.

**src/core/context.py**

class AppContext:

"""

Contient les éléments partagés de l'application :

- root : fenêtre principale

- menu : barre de menu

- current\_page : frame affichée

- chemins, config, base, etc.

"""

def \_\_init\_\_(self, root):

self.root = root

self.menu = None

self.current\_page = None

# Exemples de variables partagées

self.paths = {

"sources": "sources/",

"resultats": "resultats/",

}

# Tu peux aussi garder des références métier :

# self.db\_conn = None

# self.config = {}

**Si tu veux garder ton ancien variables\_communes.py**

Tu peux le transformer en un petit module de **constantes et chemins** (sans logique) :

# variables\_communes.py

DOSSIER\_SOURCES = "sources/"

DOSSIER\_RESULTATS = "resultats/"

FICHIER\_SOURCE = "ma\_source.xlsx"

Puis l’utiliser dans ton AppContext :

import variables\_communes as vc

self.paths = {

"sources": vc.DOSSIER\_SOURCES,

"resultats": vc.DOSSIER\_RESULTATS,

}

Cette approche te donne un **point d’entrée clair (main)**,  
un **contexte explicite pour les composants**,  
et un **fichier de variables stables** à part.

Souple, testable, et sans jamais recréer root.

Tu veux que je te montre maintenant comment le AppUI et le menu s’appuient sur ce contexte ?

Haut du formulaire

Bas du formulaire

ChatGPT peut commettre des erreurs. Il est recommandé de vérifier les informations importantes. Voir les préférences en matière de cookies.