

Deconstruyendo el Ejemplo: "Antes" vs. "Después"

Un caso de estudio práctico sobre código limpio

El "ANTES": El Script Desordenado

```
# ¡Problema! Una variable global peliculas = [] def agregar_pelicula(nombre): # Esta función depende de la variable global peliculas.append(nombre) print(f'{nombre}' agregada.) def mostrar_películas(): # Esta también depende de la variable global print("--- Mi Catálogo ---") for p in peliculas: print(p) # Lógica del menú (Uso) MEZCLADA con la lógica # de negocio (Definición) while True: opcion = input("1. Agregar 2. Ver ...") if opcion == '1': nombre = input("Nombre: ") agregar_pelicula(nombre) elif opcion == '2': mostrar_películas() # ...
```

Problema 1: La Variable Global

```
peliculas = [] (El "Pizarrón  
Público")
```

Esta lista es un "estado global". Es como un pizarrón en un pasillo público.

- 🏃 Cualquier función, en cualquier parte del código, puede pasar y modificarla.
- ❗ **Fuente de Bugs:** Si una película desaparece, ¿qué función la borró? Tienes que revisar TODO el código.
- ⚠ **Efectos Secundarios:** Las funciones `agregar_pelicula` y `mostrar_peliculas` son impredecibles; su resultado depende de un estado externo.

Problemas 2 y 3: Caos Estructural

Baja Cohesión ("Código Espagueti")

La **lógica de negocio** (las funciones `agregar_...`) está mezclada con la **lógica de la aplicación** (el menú `while True`). Si quieres cambiar el menú, puedes romper el catálogo.

Imposible de Reutilizar

¿Qué pasa si queremos dos catálogos (ej. "Terror" y "Comedia")? **No podemos**. Ambas funciones están "atadas" a la única lista global `peliculas`.

La Solución: Encapsular con una Clase

Usamos Programación Orientada a Objetos (OOP) para crear un "molde".

La Clase CatalogoPeliculas agrupará los **Datos** (la lista) y las **Funciones** (los métodos) en un solo lugar.

El "DESPUÉS": La Clase Encapsulada

```
class CatalogoPeliculas: # El "Constructor": se llama al crear un nuevo catálogo def __init__(self, nombre): self.nombre = nombre # ¡ENCAPSULADO! Esta lista es privada del objeto self.peliculas = [] # "Método": una función que pertenece a la clase def agregar_pelicula(self, nombre): self.peliculas.append(nombre) print(f'{nombre} agregada a {self.nombre}') def mostrar_peliculas(self): print(f'--- {self.nombre} ---') for p in self.peliculas: print(p) # --- La lógica de USO está SEPARADA --- catalogo_estrenos = CatalogoPeliculas("Estrenos") catalogo_estrenos.agregar_pelicula("Dune 2")
```

Ganancia 1: Encapsulación (¡Adiós Globales!)

`self.peliculas = []` (La "Libreta Privada")

La variable global ha desaparecido. Ahora:

- 🔒 La lista `peliculas` vive **DENTRO** de cada objeto (`self`).
- 🛡️ Es **privada**. No se puede acceder a ella desde fuera por accidente.
- ✓ La única forma de modificarla es usando sus propios métodos (`.agregar_pelicula()`).
- ✖️ **Control Total:** Los "efectos secundarios" se eliminan. El código es predecible.

Ganancia 2: Reutilización (¡Instancias!)

El "Molde" vs. Las "Galletas"

La `class` es el molde. Los **objetos** (o "instancias") son las galletas que creamos con él.

¡Ahora SÍ podemos tener múltiples catálogos! Cada uno es un objeto independiente con su propia lista `self.peliculas`.

```
# Usamos el MISMO molde (Clase) # para crear
# DOS objetos separados: catalogo_estrenos =
catalogo_estrenos = CatalogoPeliculas("Estrenos 2024")
catalogo_terror = CatalogoPeliculas("Clásicos
de Terror") # Esta película va a la lista de
'estrenos'
catalogo_estrenos.agregar_pelicula("Dune 2") #
Esta película va a la lista de 'terror'
catalogo_terror.agregar_pelicula("El
Exorcista") # ¡No hay forma de que se mezclen!
catalogo_estrenos.mostrar_peliculas()
catalogo_terror.mostrar_peliculas()
```

Ganancia 3: Separación de Intereses

■ Lógica de Negocio (La Clase)

```
class CatalogoPeliculas:
```

Define **QUÉ ES** un catálogo y cómo funciona internamente.
Es el "cerebro".

Es un bloque de Lego independiente.

≡ Lógica de Aplicación (El Script)

```
while True: ...
```

Define **CÓMO SE USA** el catálogo (el menú, las opciones del usuario). Es el "controlador".

Podemos cambiar el menú sin romper el "cerebro".

Comparación: ANTES vs. DESPUÉS

ANTES (Script)

-  Estado Global
-  Lógica Mezclada
-  Frágil y difícil de depurar
-  Rígido y no reutilizable

DESPUÉS (Clase)

-  Estado Encapsulado
-  Intereses Separados
-  Robusto y predecible
-  Flexible y reutilizable

Conclusión

Ambos códigos "funcionan", pero solo uno es **mantenible**.

Escribimos código limpio no para la máquina, sino para el **siguiente programador** (que a menudo eres tú mismo, 6 meses después).

¿Preguntas?

¡El código limpio es un hábito, no un evento!

¡A refactorizar!

Image Sources



https://www.juventudeinfancia.gob.es/sites/default/files/infancia/comite_expertos/INFORME_DEL_COMIT%C3%89_DE_PERSONAS_EXPERTAS_PARA_EL_DESARROLLO_DE_UN.png

Source: www.juventudeinfancia.gob.es



<https://thumbs.dreamstime.com/b/icono-de-protecci%C3%B3n-seguridad-con-bloqueo-datos-personales-escudo-cerradura-y-estrellas-ilustraci%C3%B3n-vectorial-235621614.jpg>

Source: es.dreamstime.com