Interfaces Gráficas (GUI) con Tkinter

Clase 3: Estructura (Clases) y Recursos

Prof. D.Sc. BARSEKH-ONJI Aboud

Facultad de Ingeniería
Universidad Anáhuac México

18 de noviembre de 2025

Agenda

Repaso: Nuestra App "Saludador" (Clase 2)

Nuestro Código Anterior

La app "Saludador" funciona, pero tiene un problema. Todos nuestros widgets (‘etiqueta_nombre‘, ‘entrada_nombre‘, ‘etiqueta_resultado‘, etc.) y nuestras funciones (‘saludar‘) "flotan" en el código.

El Problema del "Código Espagueti"

- ¿Qué pasa si nuestra app crece y tiene 50 botones y 30 etiquetas? Todas las variables estarían mezcladas.
- Si todo es una variable "global", es muy fácil modificar accidentalmente la variable equivocada desde la función equivocada.
- Es imposible de mantener, depurar o reutilizar.

Agenda

Concepto de Clase

La Solución: Empaquetar el Código

La Programación Orientada a Objetos (OOP) nos permite crear un "molde" o "plano" para nuestra aplicación. A este molde lo llamamos **Clase**.

La Clase como Contenedor

- Una **Clase** agrupa **datos** (variables, llamadas "atributos") y **comportamiento** (funciones, llamadas "métodos") en un solo lugar.
- En lugar de tener widgets sueltos, nuestra **Clase de Aplicación** será la dueña de todos sus botones y etiquetas.
- Esto nos da **modularidad**: podemos crear, usar y destruir nuestra aplicación como un solo bloque cohesivo.

Estructura de una App con Clases

La Estructura Básica

Así es como se ve una aplicación de Tkinter escrita profesionalmente usando una Clase:

```
1  import tkinter as tk
2
3  # 1. Creamos el ''molde'' de nuestra aplicacion
4  class MiAplicacion(tk.Frame):
5      # 2. El ''constructor'': se llama al crear la app
6      def __init__(self, master):
7          # 3. Llama al constructor de tk.Frame
8          super().__init__(master)
9          # 4. Le decimos al Frame que se ''empaquete''
10         self.pack()
11         # 5. Organizamos nuestros widgets
12         self.crear_widgets()
13
14
```

Estructura de una App con Clases

La Estructura Básica

Así es como se ve una aplicación de Tkinter escrita profesionalmente usando una Clase:

```
1
2       # 6. Metodo para crear los widgets
3       def crear_widgets(self):
4           # 'self' es el contenedor
5           self.boton = tk.Button(self, text=''Clic'')
6           self.boton.pack()
7
8   # --- Codigo principal ---
9   ventana_raiz = tk.Tk()
10  # 7. Creamos la aplicacion a partir del molde
11  app = MiAplicacion(master=ventana_raiz)
12  ventana_raiz.mainloop()
13
```

Entendiendo `self`

¿Qué es `self`?

`self` es simplemente una variable que hace referencia a la **propia instancia de la clase**. Es cómo el "molde" se refiere a sí mismo.

Entendiendo `self`

De Variable Global a Atributo `self`

`self` es la clave para organizar nuestras variables. En lugar de que un widget sea global, se convierte en un "atributo" de `self`.

Antes (Desorganizado):

```
1 def mi_funcion():
2     # Usa la variable global
3     print(entrada_nombre.get())
4
5 entrada_nombre = tk.Entry(ventana
      )
6
```

Ahora (Organizado con `self`):

```
1 class MiApp(tk.Frame):
2     def crear_widgets(self):
3         # Guardamos el widget DENTRO de self
4         self.entrada_nombre = tk.Entry(self)
5
6     def mi_funcion(self):
7         # Accedemos al widget A TRAVES de self
8         print(self.entrada_nombre.get())
9
```

Agenda

Manejo de "Data"

¿Qué es "Data"?

Nuestras aplicaciones rara vez son solo código. Necesitan **recursos** o **"data"** externa para funcionar.

- Imágenes (logos, íconos, fotos)
- Archivos de sonido
- Documentos de texto (ej. para un "Acerca de...")
- Bases de datos, archivos de configuración (JSON, XML), etc.

Manejo de "Data"

Agregando una Imagen (Logo)

El widget más común para mostrar una imagen es el mismo `tk.Label` que ya usamos, pero configurado de forma diferente.

Cómo Mostrar una Imagen

Paso 1: El Objeto `PhotoImage`

Tkinter no puede usar archivos JPG o PNG directamente. Primero, debemos cargar la imagen en un objeto especial de Tkinter llamado `PhotoImage`.

```
# 'file=' es la ruta a nuestra imagen
# (PNG o GIF son los formatos mas soportados)
logo_imagen = tk.PhotoImage(file=''logo_universidad.png'')

```

Cómo Mostrar una Imagen

Paso 2: Asignar la Imagen a un Label

Una vez que tenemos el objeto `PhotoImage`, se lo asignamos a la propiedad `image` de un `Label`.

```
# Creamos un Label, pero en lugar de 'text=',
# usamos 'image='
etiqueta_logo = tk.Label(self, image=logo_imagen)

```

Cómo Mostrar una Imagen

¡CUIDADO! El Bug del Recolector de Basura

Si creas el `PhotoImage` dentro de una función, Python lo borrará de la memoria (para "limpiar") y la imagen no se verá. **Solución:** Siempre guarda una referencia al objeto imagen, preferiblemente usando `self`.

```
self.logo_imagen = tk.PhotoImage(...)
```

Agenda

Tarea 3: Mini Calculadora v2.0 (con Clases)

Objetivo

Crear una aplicación gráfica que funcione como una sumadora simple, aplicando todo lo visto en clase.

Tarea 3: Mini Calculadora v2.0 (con Clases)

Instrucciones

- Toma tu código funcional de la Mini Calculadora de la Tarea 2.
- Crea una nueva clase.
- Mueve **todo** el código de creación de widgets (Labels, Entries, Button) al método `crear_widgets(self)`.
- Asegúrate de que todos los widgets se guarden como atributos de `self` (ej: `self.entrada_num1`, `self.etiqueta_resultado`).

Tarea 3: Mini Calculadora v2.0 (con Clases)

Instrucciones

- Mueve la función `sumar()` para que sea un **método** de la clase (ej: `def sumar(self):`).
- Actualiza el método `sumar(self)` para que lea y escriba en los widgets usando `self`.
- **(Nuevo)** Carga una imagen (¡descarga un ícono de calculadora!) y muéstrala en la parte superior de la ventana.

Tarea 3: Mini Calculadora v2.0 (con Clases)

Estructura del Código Principal

El código fuera de la clase debe ser mínimo:

```
    # ... (aqui va tu 'class Sumadora(tk.Frame):' ...)

if __name__ == "__main__":
    ventana_raiz = tk.Tk()
    ventana_raiz.title("Calculadora v2.0")
    app = Sumadora(master=ventana_raiz)
    app.mainloop()
```