Interfaces Gráficas (GUI) con Tkinter

Clase 5: Geometría de Cuadrículas y Lógica de Juegos

Prof. D.Sc. BARSEKH-ONJI Aboud

Facultad de Ingeniería
Universidad Anáhuac México

6 de noviembre de 2025

Agenda

El Límite de .pack()

Repaso: .pack()

Hasta ahora, hemos usado .pack() para posicionar nuestros widgets.

- Es simple y rápido para formularios (como la Tarea 3 de Calculadora o la Tarea 4 de Login).
- Simplemente "apila" los widgets uno encima del otro.

El Límite de .pack()

El Problema para Proyectos de Tablero

¿Qué pasa si queremos crear un calendario, un tablero de ajedrez o cualquier proyecto basado en una **cuadrícula precisa**?

- Con `.pack()`, no podemos decirle a un botón: "Ve exactamente a la fila 3, columna 5".
- Intentar crear un tablero de 10x10 con `.pack()` es casi imposible.

La Solución: .grid()

El Gestor de Geometría .grid()

Tkinter nos da una alternativa a .pack() mucho más poderosa para este tipo de tareas: .grid().

- .grid() organiza los widgets en una **tabla invisible de filas y columnas**.
- Simplemente especificamos la fila (row) y la columna (column) donde queremos que viva el widget.

La Solución: .grid()

Sintaxis Básica

```
# En lugar de .pack(), usamos .grid()
etiqueta_A = tk.Label(self, text="Celda (0, 0)")
etiqueta_A.grid(row=0, column=0)

etiqueta_B = tk.Label(self, text="Celda (0, 1)")
etiqueta_B.grid(row=0, column=1)

etiqueta_C = tk.Label(self, text="Celda (1, 0)")
etiqueta_C.grid(row=1, column=0)

etiqueta_D = tk.Label(self, text="Celda (1, 1)")
etiqueta_D.grid(row=1, column=1)

```

Creando Tableros Dinámicamente

Creando la Cuadrícula

No vamos a crear cientos de widgets a mano. Para un proyecto tipo tablero (ej. 10x10), usamos un **bucle `for` anidado**.

Creando Tableros Dinámicamente

Ejemplo: Tablero de 10x10 Botones

Esta es la técnica fundamental para crear tableros de juego:

```
# Dentro de un método, ej: crear_tablero(self)

# 1. Creamos un Frame para ser el contenedor
frame_tablero = tk.Frame(self)
frame_tablero.pack() # El frame se empaqueta

# 2. Creamos la cuadrícula DENTRO del frame
for i in range(10): # 10 filas (0 a 9)
    for j in range(10): # 10 columnas (0 a 9)
        celda = tk.Button(frame_tablero, text="~", width=2)
        celda.grid(row=i, column=j)
```

¡CUIDADO! `.pack()` vs `.grid()`

Regla de Oro

Nunca debes usar `.pack()` y `.grid()` **en el mismo contenedor** (en el mismo `Frame` o `Ventana`).

¿Por qué?

Ambos gestores de geometría "pelean" por el control del espacio y cómo calcular el tamaño de la ventana. Esto hará que tu aplicación se comporte de forma errática o directamente falle.

¡CUIDADO! `.pack()` vs `.grid()`

La Forma Correcta (Usar Contenedores)

La solución, como vimos en el código anterior, es usar `Frames` para aislar los gestores:

- Puedes usar `.pack()` para organizar los `Frames` principales (ej. 'frame_titulo.pack()', 'frame_tablero.pack()', 'frame_puntajes.pack()').
- Y luego, **dentro** de 'frame_tablero', usar `.grid()` para crear tu cuadrícula de botones.

Agenda

El Problema: ¿Qué Celda se Presionó?

El Desafío

Acabamos de crear un tablero de 10x10 (100 botones) usando un bucle `for` anidado.

El Problema: ¿Qué Celda se Presionó?

El Desafío

Acabamos de crear un tablero de 10x10 (100 botones) usando un bucle `for` anidado.

- Si asignamos el mismo `command` a todos los botones, ¿cómo sabe la función qué botón específico (fila y columna) la llamó?
- Queremos que 'on_clic_celda' reciba la fila 'i' y la columna 'j' del botón que fue presionado.

El Problema: ¿Qué Celda se Presionó?

Intento Incorrecto (¡Cuidado!)

Si intentamos esto, no funcionará:

```
# INCORRECTO:
# Esto llama la funcion 100 veces AL CREARLA
# y 'command' se asigna a 'None'
celda.config(command = self.on_clic_celda(i, j))

```

- Recuerda (Clase 2): Poner paréntesis '()' **ejecuta** la función inmediatamente.
- 'command' debe ser asignado al **nombre** de una función, no a su resultado.

La Solución: `lambda` para `command`

¿Qué es una `lambda`?

Una `lambda` es una forma rápida de definir una pequeña **función anónima** (sin nombre) en una sola línea.
La usamos para "envolver" nuestra llamada de función y "congelar" los parámetros que queremos enviarle.

La Solución: lambda para command

Sintaxis Correcta para Cuadrículas

La sintaxis clave es usar 'lambda' para crear una nueva función sobre la marcha para cada botón:

```
# Sintaxis clave:
comando_celda = lambda r=i, c=j: self.on_clic_celda(r, c)

# Asignamos esa funcion lambda al comando
celda.config(command = comando_celda)

# O todo en una linea:
celda.config(command=lambda r=i, c=j: self.on_clic_celda(r, c))

```

La Solución: `lambda` para `command`

¿Cómo Funciona?

La parte `r=i, c=j` es un truco de Python:

- "Congela" el valor **actual** de 'i' y 'j' (ej. '3' y '5') como valores por defecto para 'r' y 'c' en esa 'lambda' específica.
- Cuando el usuario hace clic, se ejecuta la 'lambda', la cual a su vez llama a nuestra función 'on_clic_celda' pasándole los valores que "recordaba" (ej. '3' y '5').

Código: Tablero Interactivo Completo

El Patrón de Diseño Completo

Así es como se ve la estructura completa dentro de tu clase de aplicación:

```
# --- Dentro de tu Clase (ej: class MiApp(tk.Frame)) ---

    # 1. El metodo que crea los widgets
    def crear_widgets(self):
        # ... (otro codigo) ...
        self.crear_tablero_juego()

    def crear_tablero_juego(self):
        frame_tablero = tk.Frame(self)
        frame_tablero.pack()

        for i in range(10): # Fila
            for j in range(10): # Columna
                celda = tk.Button(frame_tablero, text="~", width=2)
                # ¡EL PASO CLAVE!
```

Agenda

Separando Lógica y Vista

Recordatorio de la Clase 3: Clases

Nuestra clase de aplicación (ej. 'MiApp') no solo contiene los widgets (la **Vista**), sino también el estado interno del juego (la **Lógica**).

- **Vista:** La cuadrícula de botones 'tk.Button'.
- **Lógica:** Una matriz interna (lista de listas) que representa qué hay en cada celda (ej. una pieza, un número, agua, etc.).

Juegos Impredecibles

Nuestros proyectos de tablero no pueden ser siempre iguales. Necesitamos que la configuración inicial (la posición de las piezas, el valor de las cartas) sea **aleatoria** cada vez que se juega.

Separando Lógica y Vista

El Módulo `random` de Python

Python nos da una biblioteca estándar para manejar todo tipo de aleatoriedad. Primero, debemos importarla al inicio de nuestro archivo:

```
1 import tkinter as tk
2 import random # <-- ¡Importante!
3
```

Funciones Clave de `random`

1. `random.randint(a, b)`

Devuelve un número **entero** aleatorio entre ‘a‘ y ‘b‘ (ambos incluidos).

```
# Simular un dado de 6 caras
dado = random.randint(1, 6)
print(dado) # Imprime 1, 2, 3, 4, 5, o 6

# Elegir una fila aleatoria en un tablero de 10x10
fila_inicial = random.randint(0, 9)

```

Funciones Clave de `random`

2. `random.choice(lista)`

Elige un elemento al azar de una secuencia (como una lista).

```
# Elegir una orientacion para una pieza
opciones = ['horizontal', 'vertical']
orientacion = random.choice(opciones)
print(orientacion) # Imprime 'horizontal' o 'vertical'

```

Funciones Clave de `random`

3. `random.shuffle(lista)`

Esta es una de las funciones más útiles. Toma una lista y **la revuelve** en su lugar (¡no devuelve una lista nueva, modifica la original!).

Funciones Clave de `random`

Ejemplo: Revolver un Mazo de Cartas

Perfecto para juegos de emparejamiento o cartas.

```
# 1. Creamos la lista de pares
mazo = [1, 1, 2, 2, 3, 3, 4, 4]
print(f"Mazo original: {mazo}")

# 2. Revolver la lista
random.shuffle(mazo)
print(f"Mazo revuelto: {mazo}")

# Mazo revuelto: [3, 1, 4, 2, 4, 1, 3, 2] (o similar)

```

Funciones Clave de `random`

Aplicación

Al iniciar el juego, crearías tu lista de datos lógicos, la revolverías con `shuffle`, y luego usarías esa lista revuelta para asignar los valores a tu cuadrícula.

Agenda

El Problema: Pausar el Juego

El Desafío

Pensemos en un juego de emparejamiento (memoria). La secuencia de eventos es:

1. El jugador hace clic en la "Carta 1" (se voltea).
2. El jugador hace clic en la "Carta 2" (se voltea).
3. El programa verifica si son un par.
4. **Si NO son un par:** El juego debe **pausar por 1 segundo** (para que el jugador memorice) y luego volver a voltear ambas cartas.

El Problema: Pausar el Juego

El Error Más Común: `time.sleep()`

Un programador principiante intentaría usar la biblioteca `time` de Python:

```
import time

def verificar_par(self):
    # ... codigo para voltear Carta 2 ...
    if self.carta1.valor != self.carta2.valor:

        # ¡NO HAGAS ESTO!
        time.sleep(1) # Pausa por 1 segundo

        # ... codigo para esconderlas ...

```

¿Por Qué `time.sleep()` Destruye tu GUI?

Recordando el `ventana.mainloop()`

Nuestra aplicación gráfica se mantiene viva gracias a un bucle infinito llamado `mainloop()`. Este bucle hace dos cosas constantemente:

1. **Procesa eventos:** Revisa si el mouse se movió, si se hizo clic en un botón, etc.
2. **Redibuja la pantalla:** Actualiza la apariencia de los widgets.

¿Por Qué `time.sleep()` Destruye tu GUI?

El Bloqueo del Hilo Principal

- Cuando llamas a `time.sleep(1)`, le dices a Python: "¡Detén **TODO** lo que estás haciendo en este hilo por 1 segundo!".
- Esto **congela el `mainloop()` por completo**.
- Durante ese segundo, la GUI no puede procesar eventos (clics) ni redibujarse. La ventana entera aparecerá como "Congelada" o "(No Responde)".
- Nunca, bajo ninguna circunstancia, uses `time.sleep()` en el hilo principal de una GUI.

La Solución de Tkinter: `.after()`

Programación Asíncrona

La forma correcta es pedirle al `mainloop()` que ejecute una función **en el futuro**, sin detenerse ahora.
Para esto, todos los widgets (incluida la ventana raíz) tienen el método `.after()`.

La Solución de Tkinter: .after()

Sintaxis de .after()

```
# Sintaxis:
# mi_ventana.after(milisegundos, funcion_a_llamar)

# Ejemplo:
# Llama a la funcion 'self.esconder_cartas'
# despues de 1000 milisegundos (1 segundo).

def verificar_par(self):
    # ... codigo ...
    if self.carta1.valor != self.carta2.valor:

        # ¡FORMA CORRECTA!
        # El mainloop sigue corriendo normalmente...
        self.master.after(1000, self.esconder_cartas)

    else:
```

La Solución de Tkinter: `.after()`

Importante

Al igual que con `command`, se pasa el **nombre de la función** ('self.esconder_cartas') **sin paréntesis**.

Manejo de Estado (Desactivar Clics)

Un Problema Sutil

En el ejemplo anterior, ¿qué pasa si el jugador hace clic en una tercera carta *durante* el segundo de espera? ¡El juego se romperá!

Manejo de Estado (Desactivar Clics)

Solución: Variables de Estado

```
class MiApp(tk.Frame):
    def __init__(self, master):
        # ...
        # El jugador puede hacer clic al inicio
        self.permitir_clic = True
        self.crear_widgets()

    def on_clic_celda(self, r, c):
        # 1. Ignorar el clic si no esta permitido
        if not self.permitir_clic:
            return # Salir de la funcion

        # ... (logica de voltear carta) ...

        if self.dos_cartas_volteadas:
            # 2. Desactivar clics temporalmente
```

Agenda

El Problema: ¿Dónde vive el "Juego"?

El Desafío

Hemos creado una cuadrícula de botones interactivos. Pero, ¿cómo sabe el programa qué hay "debajo" de cada botón?

- El `tk.Button` es solo un widget gráfico (la **Vista**). No "sabe" si es agua, un barco, una carta de 'Rey' o una 'Reina'.
- El juego en sí (el estado, las reglas) debe vivir en una estructura de datos separada (el **Modelo**).

El Problema: ¿Dónde vive el "Juego"?

La Arquitectura Modelo-Vista-Controlador (MVC)

Nuestra Clase de Aplicación ('MiApp') actuará como las tres cosas:

- **Modelo (Datos):** Una lista de listas (matriz) que guarda el estado *real* del juego. (Ej: 'self.tablero_logico').
- **Vista (GUI):** La cuadrícula de widgets `tk.Button` que el usuario ve. (Ej: 'self.tablero_gui').
- **Controlador (Lógica):** Los métodos (como 'on_clic_celda') que conectan la Vista y el Modelo.

Paso 1: Crear el Modelo (Datos)

El Estado Lógico

En el constructor (‘__init__‘), antes de crear widgets, creamos nuestro modelo de datos.

- Para un juego 10x10, esto es una matriz (lista de listas).
- Para un juego de cartas, puede ser una lista 1D (revuelta).

Paso 1: Crear el Modelo (Datos)

Ejemplo: Modelo para un Tablero 10x10

```
# En __init__(self, master):
super().__init__(master)

# 1. Crear el MODELO
# (0 = vacio, 1 = pieza)
self.modelo_logico = []
for i in range(10):
    fila = [0] * 10 # Una fila de 10 ceros
    self.modelo_logico.append(fila)

# (Aqui iria la logica de 'random' para
#  colocar piezas, cambiando los 0 por 1)

# 2. Crear la VISTA
self.crear_widgets_gui()
```

Paso 2: Guardar la Vista (Widgets)

El Problema

Cuando el usuario hace clic en ‘(3, 4)‘, nuestro controlador (‘on_clic_celda‘) necesita:

1. Revisar ‘self.modelo_logico[3][4]‘.
2. Y luego, **actualizar** el botón específico en ‘(3, 4)‘.

¿Cómo encontramos ese botón específico entre los 100 que creamos?

Paso 2: Guardar la Vista (Widgets)

Solución: Guardar Referencias a los Widgets

Así como guardamos el modelo, también guardamos los widgets de la GUI en su propia matriz.

```
def crear_widgets_gui(self):
    frame_tablero = tk.Frame(self)
    frame_tablero.pack()

    # Matriz para guardar los botones
    self.tablero_gui = []

    for i in range(10):
        fila_gui = []
        for j in range(10):
            celda = tk.Button(frame_tablero, text="~", width=2,
                command=lambda r=i, c=j: self.on_clic_celda(r, c))
            celda.grid(row=i, column=j)
```

Paso 3: El Controlador (Conector)

El Flujo de Clic Completo

Ahora el método ‘on_clic_celda‘ puede actuar como el ”cerebro” (Controlador) que conecta todo:

Paso 3: El Controlador (Conector)

El Método Controlador

```
def on_clic_celda(self, fila, columna):
    # 1. Controlador consulta el MODELO
    valor_logico = self.modelo_logico[fila][columna]

    # 2. Controlador toma una decision
    if valor_logico == 0:
        print("Logica: Toco 'vacio'")

        # 3. Controlador actualiza la VISTA
        boton_presionado = self.tablero_gui[fila][columna]
        boton_presionado.config(text="O", bg="blue")

    elif valor_logico == 1:
        print("Logica: Toco 'pieza'")

        # 3. Controlador actualiza la VISTA
```