Manual Técnico del Código

calculadora_IntegralesDef.py

1. Visión General del Proyecto

El script calculadora_IntegralesDef.py es una **aplicación de escritorio** construida con la librería **tkinter** que actúa como una interfaz gráfica para la biblioteca de matemáticas simbólicas **sympy**. Su propósito es permitir a los usuarios calcular integrales definidas de funciones de una variable. La aplicación no solo muestra el resultado numérico, sino que también utiliza matplotlib para **renderizar la integral, el resultado y los pasos** en formato de imagen a partir de código **LaTeX**, lo que mejora la claridad y la presentación matemática.

---

2. Librerías y Dependencias

Este script requiere las siguientes bibliotecas para su correcto funcionamiento:

- **tkinter**: Es el módulo estándar de Python para crear interfaces gráficas. Se usa para construir la ventana principal (tk.Tk), botones (ttk.Button), campos de texto (ttk.Entry), etiquetas (tk.Label), y marcos (tk.Frame).
- **ttk**: Es una versión moderna de los widgets de tkinter. Se usa para dar un aspecto más actual a los botones y campos de entrada.
- **PIL (Pillow)**: La biblioteca de procesamiento de imágenes de Python. Se utiliza para abrir las imágenes generadas por matplotlib y convertirlas a un formato que tkinter pueda mostrar en sus etiquetas.
- **matplotlib.pyplot**: Se usa para crear figuras y ejes, lo que nos permite renderizar texto en formato LaTeX como una imagen.
- **re**: El módulo de expresiones regulares de Python. Es fundamental para la función de preprocesamiento, ya que permite buscar y reemplazar patrones en la cadena de texto de la función de manera eficiente.
- **sympy**: El corazón del cálculo. Proporciona las funciones symbols, sympify e integrate para definir variables, convertir cadenas de texto a expresiones matemáticas y, por supuesto, calcular integrales.
- **matplotlib.use("Agg")**: Esta línea es crucial. Agg es un backend no interactivo de matplotlib, lo que significa que el script puede generar gráficos sin necesidad de una

interfaz gráfica de usuario para mostrarlos. Esto evita que la aplicación se "congele" o genere ventanas de gráficos adicionales.

---

3. Funciones Globales

preprocesar_funcion(texto)
Esta función es vital para el **análisis de la entrada del usuario**. sympy requiere un formato de cadena muy específico para interpretar las funciones. Esta función toma una cadena de texto y la prepara, corrigiendo notaciones comunes que el usuario podría utilizar.

- texto = texto.replace(" ", ""): Elimina los espacios en blanco, simplificando el análisis de la cadena.
- texto = texto.replace("π", "pi"): Convierte el símbolo π a pi, que es como sympy lo reconoce.
- texto = texto.replace("²", "**2")...: Reemplaza los superíndices comunes ² y ³ por su notación de exponente en Python (**). Esto es esencial para que sympy interprete x^2 como x**2.
- texto = texto.replace("^", "**"): Hace lo mismo con el símbolo ^, que es una notación de exponente común.
- re.sub(r"√(\w+)", r"sqrt(\1)", texto): Utiliza una expresión regular para encontrar el símbolo de raíz cuadrada seguido de un carácter alfanumérico (p. ej., √x) y lo reemplaza con sqrt(x). La expresión r"√(\w+)" busca el √ seguido de uno o más caracteres alfanuméricos (\w+) y los captura en un grupo (). El r"sqrt(\1)" utiliza ese grupo capturado (\1) para construir la nueva cadena.
- re.sub(r"√\((.*?)\)", r"sqrt(\1)", texto): Es similar al anterior, pero busca raíces cuadradas que contienen una expresión entre paréntesis (p. ej., √(x+1)), capturando todo lo que está dentro de los paréntesis (.*?) y colocándolo dentro de sqrt().
- re.sub(r"e**(\w+)", r"exp(\1)", texto): Convierte la notación e^x a la función exp(x) que sympy puede interpretar.
- re.sub(r"(\d)([a-zA-Z\(])", r"\1*\2", texto): Este es un paso de preprocesamiento muy importante. Inserta un asterisco * para la **multiplicación implícita**. Por ejemplo, transforma "2x" en "2*x" o "3(x+1)" en "3*(x+1)". (\d) captura un dígito, y ([a-zA-Z\(]) captura una letra o un paréntesis de apertura, insertando * entre ambos.
- re.sub(r"(\))(\w)", r"\1*\2", texto): Similar al anterior, inserta un * entre un paréntesis de cierre y una variable, por ejemplo, transformando "(x+1)y" en "(x+1)*y".

---

4. Clase `CalculadoraIntegralLatex`

Esta es la clase principal que maneja la interfaz de usuario y la lógica de la aplicación. Hereda de `tk.Tk` para funcionar como la ventana principal.

`__init__(self)`

El constructor de la clase inicializa la ventana y sus componentes.

- `super().__init__()`: Llama al constructor de la clase padre (`tk.Tk`).
- `self.title(...)` y `self.geometry(...)`: Establece el título y el tamaño inicial de la ventana.
- `self.funcion = tk.StringVar(...)`: `tk.StringVar` es una clase de variable de control de `tkinter` que se usa para enlazar el contenido de un widget (`Entry`) con una variable de Python. Esto permite que el código acceda o modifique fácilmente el texto del campo de entrada. Las variables `self.limite_a`, `self.limite_b` y `self.resultado` funcionan de manera similar.
- `self.locals_dict`: Un diccionario que mapea nombres de funciones y constantes comunes (`sin`, `cos`, `pi`) a sus respectivos objetos de `sympy`. Se pasa a `sympify` para que pueda reconocerlos.
- `self.style = ttk.Style()`: Crea un objeto de estilo para personalizar la apariencia de los widgets de `ttk`, como los botones.
- `self.construir_ui()`: Llama al método encargado de la construcción de la interfaz.

`construir_ui(self)`

Este método se encarga de la **disposición visual de la aplicación**.

- `frame_main = tk.Frame(...)`: Un `Frame` es un contenedor que se usa para agrupar otros widgets. Esto ayuda a organizar la interfaz.
- `tk.Label(...)`: Muestra texto estático. El `font` y los colores se configuran para mejorar la estética.
- `ttk.Entry(...)`: Un campo de entrada de texto. El parámetro `textvariable=self.funcion` vincula este widget a la variable de control `self.funcion`.
- `for texto, valor in botones:`: Este bucle crea los botones de funciones predefinidas de forma dinámica. La función `lambda val=valor: self.agregar(val)` crea una función anónima que "congela" el valor actual de `valor` en el momento de la creación del botón, evitando un problema común en los bucles (`late-binding`).
- **Sección de los pasos**:
- `self.frame_pasos`: Un `Frame` que sirve como contenedor para el área de los pasos.
- `self.canvas`: Un `Canvas` es un widget que se puede usar para dibujar o, en este caso, para albergar otros widgets en un área con desplazamiento.
- `self.v_scroll`: Un `Scrollbar` que se enlaza al `Canvas` para permitir el desplazamiento vertical.

- self.inner_frame: Un Frame que vive dentro del Canvas. Todos los widgets de los pasos se colocan aquí. El Canvas mostrará y permitirá el desplazamiento de este Frame.
- self.inner_frame.bind(...): Este bind es fundamental. Cada vez que el tamaño del inner_frame cambia (por ejemplo, cuando se agrega un nuevo paso), esta línea ajusta la región de desplazamiento del Canvas para que la barra de desplazamiento funcione correctamente.
  agregar(self, valor)
  Este es un método sencillo que simplemente agrega una cadena de texto al final del contenido del campo de entrada de la función.

  calcular(self)
  Esta es la **lógica principal de cálculo**.
- try...except: Se utiliza para manejar posibles errores, como la entrada de una función no válida (SympifyError) o límites de integración no numéricos (ValueError).
- x = symbols("x"): Define la variable x como un símbolo matemático, lo que permite a sympy realizar operaciones simbólicas con ella.
- texto = preprocesar_funcion(...): Llama a la función de preprocesamiento para limpiar y preparar la entrada.
- f_sym = sympify(texto, locals=self.locals_dict): Convierte la cadena de texto preprocesada en un objeto de expresión simbólica.
- a_expr = sympify(...), b_expr = sympify(...): Permite que los límites de integración sean expresiones simbólicas (p. ej., pi/2) antes de ser convertidos a números flotantes.
- F = integrate(f_sym, x): Calcula la **integral indefinida** de la función. El resultado F es la antiderivada.
- resultado_integral = integrate(f_sym, (x, a, b)): Calcula la **integral definida** de la función entre los límites a y b.
- valor_decimal = float(resultado_integral.evalf()): evalf() evalúa la expresión simbólica para obtener un valor numérico de punto flotante.
- integral_latex = ...: Construye la cadena LaTeX para el resultado final, combinando la notación de integral con los límites y los resultados. latex(f_sym) convierte la expresión de sympy a formato LaTeX.
- self.render_latex(...): Muestra la imagen del resultado principal.
- self.mostrar_pasos(...): Llama al método para generar y mostrar los pasos detallados del cálculo.
  mostrar_pasos(self, ...)

Este método genera la secuencia de pasos de la integración definida, basándose en el **Teorema Fundamental del Cálculo**.

- `for widget in self.inner_frame.winfo_children(): widget.destroy()`: Limpia el contenido del `Frame` de los pasos antes de agregar los nuevos.
- `pasos = [...]`: Una lista de cadenas LaTeX que representan los pasos.
- `rf"\int_{{{a}}}^{{{b}}} {latex(f_sym)}\,dx"`: Muestra la integral original.
- `rf"F(x) = {latex(F)}"`: Muestra la antiderivada.
- `rf"F({b}) - F({a})"`: Muestra la evaluación simbólica.
- `rf"{latex(F.subs('x', b))}"`: `F.subs('x', b)` sustituye simbólicamente la variable `x` por el límite superior `b` en la antiderivada `F`. Esto se usa para mostrar la evaluación de la función en cada límite.
- `for i, paso in enumerate(pasos, start=1)`: Un bucle que itera sobre la lista de pasos, asignando un número a cada uno para la visualización.

  agregar_paso_latex(self, latex_expr)

  Esta función es la encargada de **renderizar una sola línea de LaTeX** como una imagen y mostrarla en la interfaz.
- `plt.figure(...)`: Crea una figura de `matplotlib`.
- `plt.text(...)`: Dibuja el texto en formato LaTeX dentro de la figura. El `$latex_expr$` indica que la cadena debe ser interpretada como LaTeX.
- `plt.axis("off")`: Desactiva los ejes de la figura para que solo se vea el texto.
- `plt.savefig(...)`: Guarda la figura como un archivo PNG temporal. `dpi=150` mejora la calidad de la imagen, y `transparent=True` elimina el fondo blanco.
- `Image.open(...)`, `ImageTk.PhotoImage(...)`: Usa Pillow para abrir la imagen PNG guardada y la convierte en un formato compatible con `tkinter`.
- `tk.Label(...)`: Crea una nueva etiqueta de `tkinter` y le asigna la imagen convertida.
- `label_img.pack(...)`: Muestra la etiqueta dentro del `inner_frame` del área de desplazamiento.

  render_latex(self, latex_expr)

  Funciona de manera similar a `agregar_paso_latex`, pero está optimizada para mostrar el **resultado principal**. Utiliza una figura más grande y un tamaño de fuente mayor para que el resultado destaque. En lugar de crear una nueva etiqueta, actualiza la imagen de una etiqueta preexistente (`self.imagen_latex`), lo cual es más eficiente.

---

5. Ejecución del Programa

- `if __name__ == "__main__":`: Esta condición es estándar en Python. Asegura que el código dentro de ella solo se ejecute cuando el script es el programa principal.
- `app = CalculadoraIntegralLatex()`: Crea una instancia de la clase principal, lo que inicializa la ventana.
- `app.mainloop()`: Inicia el bucle de eventos de `tkinter`. Este bucle espera eventos (clics, pulsaciones de teclas) y ejecuta el código correspondiente. Sin esta línea, la ventana se cerraría inmediatamente después de ser creada.