Tutorial Tkinter

José Alfredo F. Costa – Setembro de 2024

UFRN FAP 2024

1. Introdução ao Tkinter

1.1 O que é Tkinter?

Tkinter é a biblioteca padrão de interface gráfica do usuário (GUI) para Python. O nome "Tkinter" vem de "Tk interface", pois é uma interface Python para o kit de ferramentas Tk, originalmente desenvolvido para a linguagem Tcl.

Principais características do Tkinter:

- Vem pré-instalado com a maioria das distribuições Python
- Oferece uma abordagem simples e rápida para criar GUIs
- Multiplataforma, funcionando em Windows, macOS e Linux
- Leve e adequado para projetos pequenos a médios

1.2 Por que usar Tkinter?

Existem várias razões para escolher Tkinter para desenvolvimento de GUIs em Python:

1. **Facilidade de uso**: Tkinter tem uma curva de aprendizado suave, tornando-o ideal para iniciantes.
2. **Padrão da biblioteca Python**: Não requer instalação adicional na maioria dos casos.
3. **Rapidez no desenvolvimento**: Permite criar interfaces funcionais com poucas linhas de código.
4. **Portabilidade**: Aplicativos Tkinter funcionam em diferentes sistemas operacionais sem modificações.
5. **Documentação abrangente**: Ampla disponibilidade de recursos de aprendizado e comunidade ativa.

1.3 Alternativas ao Tkinter

Embora Tkinter seja uma excelente escolha para muitos projetos, existem alternativas que podem ser mais adequadas em certos casos:

1. **PyQt/PySide**: Oferece widgets mais modernos e recursos avançados, mas tem uma curva de aprendizado mais íngreme.
2. **wxPython**: Proporciona uma aparência nativa em diferentes plataformas, mas pode ser mais complexo de usar.
3. **Kivy**: Excelente para aplicativos multitoque e com foco em dispositivos móveis.
4. **PyGObject (GTK)**: Bom para criar aplicativos com aparência nativa no Linux.

Cada alternativa tem seus prós e contras, e a escolha depende das necessidades específicas do projeto.

2. Configuração do Ambiente

2.1 Instalação do Python

Para usar o Tkinter, primeiro precisamos ter o Python instalado. Siga estes passos:

1. Acesse o site oficial do Python (https://www.python.org)
2. Baixe a versão mais recente do Python para seu sistema operacional
3. Execute o instalador e siga as instruções na tela
4. Certifique-se de marcar a opção "Add Python to PATH" durante a instalação

Para verificar se a instalação foi bem-sucedida, abra um terminal ou prompt de comando e digite:

```
python --version
```

Isso deve exibir a versão do Python instalada.

2.2 Verificação da instalação do Tkinter

O Tkinter geralmente vem pré-instalado com o Python. Para verificar se está disponível, você pode usar o seguinte código:

```
import tkinter as tk
root = tk.Tk()
root.title("Teste Tkinter")
label = tk.Label(root, text="Olá, Tkinter!")
label.pack()
root.mainloop()
```

Se uma janela aparecer com o texto "Olá, Tkinter!", a instalação foi bem-sucedida.

Caso ocorra um erro, você pode tentar instalar o Tkinter separadamente:

- No Windows: `pip install tk`
- No macOS: `pip install tk`
- No Linux: `sudo apt-get install python3-tk` (para distribuições baseadas em Debian/Ubuntu)

3. Conceitos Básicos do Tkinter

3.1 Janela principal

A janela principal é o contêiner de nível superior para todos os outros widgets em um aplicativo Tkinter. Ela é criada usando a classe `Tk()`.

Exemplo básico:

```
import tkinter as tk

root = tk.Tk()
root.title("Minha Primeira Janela")
root.geometry("300x200")  # Define o tamanho da janela (largura x
altura)

root.mainloop()
```

Neste exemplo:

- Criamos uma instância de `Tk()` chamada `root`
- Definimos o título da janela com `title()`
- Configuramos o tamanho inicial da janela com `geometry()`
- Iniciamos o loop principal de eventos com `mainloop()`

3.2 Widgets

Widgets são os elementos de interface do usuário que compõem a GUI. Tkinter oferece uma variedade de widgets, como botões, rótulos, caixas de entrada, etc.

Exemplo com alguns widgets básicos:

```
import tkinter as tk

root = tk.Tk()
root.title("Widgets Básicos")

# Rótulo (Label)
label = tk.Label(root, text="Olá, sou um rótulo!")
label.pack()

# Botão (Button)
def on_button_click():
    print("Botão clicado!")

button = tk.Button(root, text="Clique-me", command=on_button_click)
button.pack()

# Entrada de texto (Entry)
entry = tk.Entry(root)
entry.pack()

root.mainloop()
```

Neste exemplo, criamos três widgets comuns:

- Um `Label` para exibir texto
- Um `Button` que imprime uma mensagem quando clicado
- Um `Entry` para entrada de texto

3.3 Geometria e layout

Tkinter oferece três principais gerenciadores de geometria para organizar widgets:

1. **pack()**: Organiza widgets em blocos antes de colocá-los no pai
2. **grid()**: Organiza widgets em uma estrutura de tabela
3. **place()**: Permite posicionar widgets usando coordenadas absolutas

Exemplo usando `grid()`:

```
import tkinter as tk

root = tk.Tk()
root.title("Layout com Grid")

# Rótulos
```

```
tk.Label(root, text="Nome:").grid(row=0, column=0, sticky="e")
tk.Label(root, text="Email:").grid(row=1, column=0, sticky="e")

# Entradas
nome_entry = tk.Entry(root)
nome_entry.grid(row=0, column=1)
email_entry = tk.Entry(root)
email_entry.grid(row=1, column=1)

# Botão
tk.Button(root, text="Enviar").grid(row=2, column=0, columnspan=2)

root.mainloop()
```

Neste exemplo:

- Usamos `grid()` para organizar os widgets em linhas e colunas
- `sticky="e"` alinha os rótulos à direita (east)
- `columnspan=2` faz o botão ocupar duas colunas

Cada gerenciador de geometria tem suas vantagens e é adequado para diferentes tipos de layouts. `pack()` é simples para layouts básicos, `grid()` é poderoso para layouts complexos em forma de grade, e `place()` oferece controle preciso sobre o posicionamento.

4. Widgets Principais do Tkinter

Nesta seção, vamos explorar os widgets mais comuns do Tkinter, suas propriedades e como usá-los efetivamente.

4.1 Label

O widget Label é usado para exibir texto ou imagens.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Label")

# Label simples
label1 = tk.Label(root, text="Olá, Mundo!")
label1.pack()

# Label com fonte personalizada
label2 = tk.Label(root, text="Label Estilizado", font=("Arial", 16,
"bold"), fg="blue")
label2.pack()

# Label com imagem
```

```
imagem = tk.PhotoImage(file="python_logo.png")
label3 = tk.Label(root, image=imagem)
label3.image = imagem  # Mantenha uma referência!
label3.pack()

root.mainloop()
```

Neste exemplo, criamos três tipos diferentes de Labels: um simples, um com estilo personalizado e um com uma imagem.

4.2 Button

O widget Button permite que o usuário acione uma ação quando clicado.

```
import tkinter as tk
from tkinter import messagebox

root = tk.Tk()
root.title("Exemplo de Button")

def clique_simples():
    messagebox.showinfo("Clique", "Botão clicado!")

def clique_com_parametro(mensagem):
    messagebox.showinfo("Clique", mensagem)

# Botão simples
btn1 = tk.Button(root, text="Clique-me", command=clique_simples)
btn1.pack()

# Botão com parâmetro usando lambda
btn2 = tk.Button(root, text="Botão com Parâmetro",
                 command=lambda: clique_com_parametro("Botão com
parâmetro clicado!"))
btn2.pack()

root.mainloop()
```

Este exemplo mostra como criar botões e associá-los a funções, incluindo o uso de lambdas para passar parâmetros.

4.3 Entry

O widget Entry é usado para entrada de texto de linha única.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Entry")

def mostrar_texto():
    texto = entrada.get()
```

```
    label_resultado.config(text=f"Você digitou: {texto}")

# Criar e posicionar o widget Entry
entrada = tk.Entry(root, width=30)
entrada.pack()

# Botão para mostrar o texto digitado
btn_mostrar = tk.Button(root, text="Mostrar Texto",
command=mostrar_texto)
btn_mostrar.pack()

# Label para exibir o resultado
label_resultado = tk.Label(root, text="")
label_resultado.pack()

root.mainloop()
```

Este exemplo demonstra como usar o Entry para receber input do usuário e exibir o texto digitado.

4.4 Text

O widget Text é usado para entrada e exibição de texto multilinha.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Text")

# Criar widget Text
texto = tk.Text(root, height=10, width=40)
texto.pack()

# Inserir texto inicial
texto.insert(tk.END, "Este é um widget Text.\n")
texto.insert(tk.END, "Você pode escrever várias linhas aqui.\n")

# Função para obter o texto
def obter_texto():
    conteudo = texto.get("1.0", tk.END)
    print("Conteúdo do Text:")
    print(conteudo)

# Botão para obter o texto
btn_obter = tk.Button(root, text="Obter Texto", command=obter_texto)
btn_obter.pack()

root.mainloop()
```

Este exemplo mostra como criar um widget Text, inserir texto nele e recuperar seu conteúdo.

4.5 Checkbutton

O Checkbutton é usado para opções que podem ser ativadas ou desativadas.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Checkbutton")

# Variáveis para armazenar o estado dos Checkbuttons
var1 = tk.IntVar()
var2 = tk.IntVar()

# Função para mostrar o estado dos Checkbuttons
def mostrar_estado():
    print(f"Opção 1: {'Marcada' if var1.get() else 'Desmarcada'}")
    print(f"Opção 2: {'Marcada' if var2.get() else 'Desmarcada'}")

# Criar Checkbuttons
cb1 = tk.Checkbutton(root, text="Opção 1", variable=var1)
cb1.pack()

cb2 = tk.Checkbutton(root, text="Opção 2", variable=var2)
cb2.pack()

# Botão para mostrar o estado
btn = tk.Button(root, text="Mostrar Estado", command=mostrar_estado)
btn.pack()

root.mainloop()
```

Este exemplo demonstra como criar Checkbuttons e verificar seu estado.

5. Layouts e Gerenciadores de Geometria

Tkinter oferece três principais gerenciadores de geometria para organizar widgets na interface.

5.1 pack()

O método `pack()` organiza widgets em blocos antes de colocá-los no widget pai.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de pack()")

# Criando frames para demonstrar o pack
frame_topo = tk.Frame(root, bg="red")
frame_meio = tk.Frame(root, bg="green")
frame_baixo = tk.Frame(root, bg="blue")
```

```
# Empacotando os frames
frame_topo.pack(fill=tk.X)
frame_meio.pack(fill=tk.BOTH, expand=True)
frame_baixo.pack(fill=tk.X)

# Adicionando alguns widgets aos frames
tk.Label(frame_topo, text="Topo", bg="red").pack()
tk.Label(frame_meio, text="Meio", bg="green").pack(expand=True)
tk.Label(frame_baixo, text="Baixo", bg="blue").pack()

root.geometry("300x200")
root.mainloop()
```

Este exemplo mostra como usar `pack()` para organizar widgets verticalmente.

5.2 grid()

O método `grid()` organiza widgets em uma estrutura de tabela.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de grid()")

# Criando widgets
label1 = tk.Label(root, text="Nome:")
entry1 = tk.Entry(root)
label2 = tk.Label(root, text="Email:")
entry2 = tk.Entry(root)
button = tk.Button(root, text="Enviar")

# Organizando widgets com grid
label1.grid(row=0, column=0, sticky="e", padx=5, pady=5)
entry1.grid(row=0, column=1, padx=5, pady=5)
label2.grid(row=1, column=0, sticky="e", padx=5, pady=5)
entry2.grid(row=1, column=1, padx=5, pady=5)
button.grid(row=2, column=0, columnspan=2, pady=10)

root.mainloop()
```

Este exemplo demonstra como usar `grid()` para criar um layout em forma de grade.

5.3 place()

O método `place()` permite posicionar widgets usando coordenadas absolutas ou relativas.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de place()")
```

```
root.geometry("300x200")

# Criando e posicionando widgets
label1 = tk.Label(root, text="Posição Absoluta", bg="yellow")
label1.place(x=50, y=20)

label2 = tk.Label(root, text="Posição Relativa", bg="lightblue")
label2.place(relx=0.5, rely=0.5, anchor="center")

button = tk.Button(root, text="Botão")
button.place(relx=1.0, rely=1.0, anchor="se")

root.mainloop()
```

Este exemplo mostra como usar `place()` para posicionar widgets com precisão.

6. Menus e Barras de Ferramentas

Menus e barras de ferramentas são elementos importantes para criar interfaces de usuário mais complexas e funcionais.

6.1 Menu

O widget Menu permite criar barras de menu e menus suspensos.

```
import tkinter as tk
from tkinter import messagebox

root = tk.Tk()
root.title("Exemplo de Menu")

# Função para as opções do menu
def acao_menu(opcao):
    messagebox.showinfo("Ação do Menu", f"Você selecionou: {opcao}")

# Criando a barra de menu
menubar = tk.Menu(root)

# Criando o menu Arquivo
filemenu = tk.Menu(menubar, tearoff=0)
filemenu.add_command(label="Novo", command=lambda: acao_menu("Novo"))
filemenu.add_command(label="Abrir", command=lambda:
acao_menu("Abrir"))
filemenu.add_separator()
filemenu.add_command(label="Sair", command=root.quit)
menubar.add_cascade(label="Arquivo", menu=filemenu)

# Criando o menu Editar
editmenu = tk.Menu(menubar, tearoff=0)
editmenu.add_command(label="Cortar", command=lambda:
acao_menu("Cortar"))
```

```
editmenu.add_command(label="Copiar", command=lambda:
acao_menu("Copiar"))
editmenu.add_command(label="Colar", command=lambda:
acao_menu("Colar"))
menubar.add_cascade(label="Editar", menu=editmenu)

# Configurando a janela para usar o menubar
root.config(menu=menubar)

root.mainloop()
```

Este exemplo cria uma barra de menu com dois menus suspensos: "Arquivo" e "Editar".

6.2 Menubutton

O Menubutton é um botão que, quando clicado, exibe um menu suspenso.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Menubutton")

# Criando o Menubutton
mb = tk.Menubutton(root, text="Opções", relief=tk.RAISED)
mb.pack(padx=10, pady=10)

# Criando o menu para o Menubutton
mb_menu = tk.Menu(mb, tearoff=0)
mb["menu"] = mb_menu

# Variáveis para as opções do menu
opcao1_var = tk.IntVar()
opcao2_var = tk.IntVar()

# Adicionando opções ao menu
mb_menu.add_checkbutton(label="Opção 1", variable=opcao1_var)
mb_menu.add_checkbutton(label="Opção 2", variable=opcao2_var)

# Função para mostrar as opções selecionadas
def mostrar_selecao():
    print(f"Opção 1: {'Selecionada' if opcao1_var.get() else 'Não
selecionada'}")
    print(f"Opção 2: {'Selecionada' if opcao2_var.get() else 'Não
selecionada'}")

# Botão para mostrar a seleção
tk.Button(root, text="Mostrar Seleção",
command=mostrar_selecao).pack(pady=10)

root.mainloop()
```

Este exemplo demonstra como criar um Menubutton com opções de menu que podem ser marcadas ou desmarcadas.

6.3 OptionMenu

O OptionMenu é um widget que permite ao usuário escolher uma opção de uma lista suspensa.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de OptionMenu")

# Lista de opções
opcoes = ["Opção 1", "Opção 2", "Opção 3", "Opção 4"]

# Variável para armazenar a opção selecionada
opcao_selecionada = tk.StringVar(root)
opcao_selecionada.set(opcoes[0])  # valor padrão

# Criando o OptionMenu
om = tk.OptionMenu(root, opcao_selecionada, *opcoes)
om.pack(pady=10)

# Função para mostrar a opção selecionada
def mostrar_selecao():
    print(f"Opção selecionada: {opcao_selecionada.get()}")

# Botão para mostrar a seleção
tk.Button(root, text="Mostrar Seleção",
command=mostrar_selecao).pack(pady=10)

root.mainloop()
```

Este exemplo mostra como criar um OptionMenu com uma lista de opções e como recuperar a opção selecionada.

7. Widgets de Contêiner

Os widgets de contêiner são usados para agrupar e organizar outros widgets, ajudando a criar layouts mais complexos e estruturados.

7.1 Frame

O Frame é um contêiner simples usado para agrupar outros widgets.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Frame")
```

```
# Criando frames
frame1 = tk.Frame(root, bg="lightblue", padx=10, pady=10)
frame1.pack(side="left", fill="both", expand=True)

frame2 = tk.Frame(root, bg="lightgreen", padx=10, pady=10)
frame2.pack(side="right", fill="both", expand=True)

# Adicionando widgets aos frames
tk.Label(frame1, text="Frame 1").pack()
tk.Button(frame1, text="Botão 1").pack()

tk.Label(frame2, text="Frame 2").pack()
tk.Button(frame2, text="Botão 2").pack()

root.geometry("300x200")
root.mainloop()
```

Este exemplo demonstra como usar Frames para organizar widgets em grupos distintos.

7.2 LabelFrame

O LabelFrame é semelhante ao Frame, mas inclui uma borda e um título.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de LabelFrame")

# Criando LabelFrames
lf1 = tk.LabelFrame(root, text="Informações Pessoais", padx=10,
pady=10)
lf1.pack(padx=10, pady=10, fill="both", expand=True)

lf2 = tk.LabelFrame(root, text="Preferências", padx=10, pady=10)
lf2.pack(padx=10, pady=10, fill="both", expand=True)

# Adicionando widgets aos LabelFrames
tk.Label(lf1, text="Nome:").grid(row=0, column=0)
tk.Entry(lf1).grid(row=0, column=1)

tk.Label(lf1, text="Idade:").grid(row=1, column=0)
tk.Entry(lf1).grid(row=1, column=1)

tk.Checkbutton(lf2, text="Receber newsletter").pack()
tk.Checkbutton(lf2, text="Aceitar termos").pack()

root.geometry("300x200")
root.mainloop()
```

Este exemplo mostra como usar LabelFrames para criar seções rotuladas em sua interface.

7.3 PanedWindow

O PanedWindow permite criar layouts divididos que o usuário pode redimensionar.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de PanedWindow")

# Criando PanedWindow
pw = tk.PanedWindow(orient=tk.HORIZONTAL)
pw.pack(fill=tk.BOTH, expand=True)

# Criando frames para adicionar ao PanedWindow
left_frame = tk.Frame(pw, background="lightblue", width=100,
height=200)
right_frame = tk.Frame(pw, background="lightgreen", width=200,
height=200)

# Adicionando frames ao PanedWindow
pw.add(left_frame)
pw.add(right_frame)

# Adicionando conteúdo aos frames
tk.Label(left_frame, text="Painel Esquerdo").pack(pady=10)
tk.Label(right_frame, text="Painel Direito").pack(pady=10)

root.geometry("300x200")
root.mainloop()
```

Este exemplo demonstra como criar um layout dividido usando PanedWindow.

7.4 Notebook (ttk)

O Notebook permite criar interfaces com abas.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.title("Exemplo de Notebook")

# Criando Notebook
notebook = ttk.Notebook(root)
notebook.pack(fill=tk.BOTH, expand=True)

# Criando frames para as abas
tab1 = ttk.Frame(notebook)
tab2 = ttk.Frame(notebook)
tab3 = ttk.Frame(notebook)

# Adicionando frames como abas ao Notebook
```

```
notebook.add(tab1, text="Aba 1")
notebook.add(tab2, text="Aba 2")
notebook.add(tab3, text="Aba 3")

# Adicionando conteúdo às abas
tk.Label(tab1, text="Conteúdo da Aba 1").pack(padx=10, pady=10)
tk.Button(tab2, text="Botão na Aba 2").pack(padx=10, pady=10)
tk.Entry(tab3).pack(padx=10, pady=10)

root.geometry("300x200")
root.mainloop()
```

Este exemplo mostra como criar uma interface com múltiplas abas usando o widget Notebook.

8. Diálogos e Janelas Pop-up

Diálogos e janelas pop-up são úteis para interagir com o usuário, exibir mensagens ou solicitar informações adicionais.

8.1 MessageBox

O MessageBox é usado para exibir mensagens, avisos ou erros ao usuário.

```
import tkinter as tk
from tkinter import messagebox

root = tk.Tk()
root.title("Exemplo de MessageBox")

def mostrar_info():
    messagebox.showinfo("Informação", "Esta é uma mensagem
informativa.")

def mostrar_aviso():
    messagebox.showwarning("Aviso", "Este é um aviso!")

def mostrar_erro():
    messagebox.showerror("Erro", "Ocorreu um erro!")

def confirmar():
    resultado = messagebox.askquestion("Confirmação", "Deseja
continuar?")
    print(f"Resultado: {resultado}")

tk.Button(root, text="Mostrar Info",
command=mostrar_info).pack(pady=5)
tk.Button(root, text="Mostrar Aviso",
command=mostrar_aviso).pack(pady=5)
tk.Button(root, text="Mostrar Erro",
command=mostrar_erro).pack(pady=5)
```

```
tk.Button(root, text="Confirmar", command=confirmar).pack(pady=5)

root.mainloop()
```

Este exemplo demonstra diferentes tipos de MessageBox para interagir com o usuário.

8.2 FileDialog

O FileDialog permite que o usuário selecione arquivos ou diretórios.

```
import tkinter as tk
from tkinter import filedialog

root = tk.Tk()
root.title("Exemplo de FileDialog")

def abrir_arquivo():
    arquivo = filedialog.askopenfilename(
        title="Selecione um arquivo",
        filetypes=(("Arquivos de texto", "*.txt"), ("Todos os
arquivos", "*.*"))
    )
    if arquivo:
        print(f"Arquivo selecionado: {arquivo}")

def salvar_arquivo():
    arquivo = filedialog.asksaveasfilename(
        title="Salvar arquivo como",
        defaultextension=".txt",
        filetypes=(("Arquivos de texto", "*.txt"), ("Todos os
arquivos", "*.*"))
    )
    if arquivo:
        print(f"Arquivo para salvar: {arquivo}")

tk.Button(root, text="Abrir Arquivo",
command=abrir_arquivo).pack(pady=5)
tk.Button(root, text="Salvar Arquivo",
command=salvar_arquivo).pack(pady=5)

root.mainloop()
```

Este exemplo mostra como usar FileDialog para abrir e salvar arquivos.

8.3 ColorChooser

O ColorChooser permite que o usuário selecione uma cor.

```
import tkinter as tk
from tkinter import colorchooser

root = tk.Tk()
```

```
root.title("Exemplo de ColorChooser")

def escolher_cor():
    cor = colorchooser.askcolor(title="Escolha uma cor")
    if cor[1]:  # cor é uma tupla (RGB, hexadecimal)
        root.configure(bg=cor[1])
        label_cor.config(text=f"Cor selecionada: {cor[1]}")

tk.Button(root, text="Escolher Cor",
command=escolher_cor).pack(pady=20)
label_cor = tk.Label(root, text="Nenhuma cor selecionada")
label_cor.pack(pady=10)

root.geometry("300x200")
root.mainloop()
```

Este exemplo demonstra como usar o ColorChooser para selecionar uma cor e aplicá-la à janela.

9. Eventos e Vinculações

Eventos e vinculações permitem que sua aplicação responda a ações do usuário, como cliques do mouse ou pressionamentos de teclas.

```
import tkinter as tk

root = tk.Tk()
root.title("Exemplo de Eventos e Vinculações")

def on_enter(event):
    event.widget.config(bg="lightblue")

def on_leave(event):
    event.widget.config(bg="SystemButtonFace")

def on_click(event):
    print(f"Clicou em: {event.widget.cget('text')}")

def on_key(event):
    print(f"Tecla pressionada: {event.char}")

# Criando botões e vinculando eventos
for i in range(3):
    btn = tk.Button(root, text=f"Botão {i+1}")
    btn.pack(pady=5)
    btn.bind("<Enter>", on_enter)
    btn.bind("<Leave>", on_leave)
    btn.bind("<Button-1>", on_click)

# Vinculando evento de teclado à janela principal
root.bind("<Key>", on_key)
```

```
root.geometry("200x150")
root.mainloop()
```

Este exemplo demonstra como vincular eventos de mouse e teclado a widgets e à janela principal.

10. Estilização e Temas

A estilização permite personalizar a aparência dos widgets, enquanto os temas oferecem uma maneira consistente de alterar o visual da aplicação.

10.1 Configuração de estilos

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.title("Exemplo de Estilos")

# Criando um estilo
style = ttk.Style()

# Configurando estilos para diferentes widgets
style.configure("TButton", foreground="blue", font=("Arial", 10,
"bold"))
style.configure("TLabel", background="lightgray", padding=5)
style.configure("Custom.TEntry", fieldbackground="lightyellow")

# Usando os estilos
ttk.Button(root, text="Botão Estilizado").pack(pady=10)
ttk.Label(root, text="Label Estilizado").pack(pady=10)
ttk.Entry(root, style="Custom.TEntry").pack(pady=10)

root.geometry("250x200")
root.mainloop()
```

Este exemplo mostra como criar e aplicar estilos personalizados a widgets ttk.

10.2 Uso de temas

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.title("Exemplo de Temas")

# Obtendo temas disponíveis
temas_disponiveis = ttk.Style().theme_names()

# Função para mudar o tema
def mudar_tema():
```

```
    tema_atual = tema_var.get()
    ttk.Style().theme_use(tema_atual)

# Criando widgets
frame = ttk.Frame(root, padding="10")
frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

ttk.Label(frame, text="Selecione um tema:").grid(row=0, column=0,
pady=5)

tema_var = tk.StringVar()
tema_combobox = ttk.Combobox(frame, textvariable=tema_var,
values=temas_disponiveis)
tema_combobox.grid(row=1, column=0, pady=5)
tema_combobox.set(ttk.Style().theme_use())

ttk.Button(frame, text="Aplicar Tema", command=mudar_tema).grid(row=2,
column=0, pady=10)

# Alguns widgets para demonstração
ttk.Entry(frame).grid(row=3, column=0, pady=5)
ttk.Checkbutton(frame, text="Opção 1").grid(row=4, column=0, pady=5)
ttk.Radiobutton(frame, text="Opção 2").grid(row=5, column=0, pady=5)

root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)
frame.columnconfigure(0, weight=1)

root.geometry("300x300")
root.mainloop()
```

Este exemplo demonstra como mudar dinamicamente entre os temas disponíveis no Tkinter.

Estas seções fornecem uma visão abrangente dos widgets de contêiner, diálogos, eventos e estilização no Tkinter. Alguns pontos adicionais que podem enriquecer seu aprendizado:

1. **Combinação de Widgets**: Tente combinar diferentes widgets de contêiner para criar layouts mais complexos e organizados.
2. **Personalização Avançada**: Lembre que é possível criar temas personalizados completamente novos, além de usar os temas predefinidos.
3. **Boas Práticas de UI/UX**: Lembre da importância de uma interface de usuário intuitiva e como os diálogos e janelas pop-up devem ser usados com moderação.
4. **Tratamento de Erros**: Ao trabalhar com arquivos e diálogos, enfatize a importância de lidar com possíveis erros ou cancelamentos por parte do usuário.
5. **Acessibilidade**: Estude os conceitos de criação de interfaces acessíveis, com uso de certos widgets e práticas podem melhorar a acessibilidade.
6. **Internacionalização**: Lembre de, em casos de trabalhos mais avançados, de preparar a interface para suportar múltiplos idiomas.

7. **Projeto Prático**: Busque aplicar sempre em projetos que incorpore vários dos conceitos discutidos, como um editor de texto simples com múltiplas abas, diálogos para abrir/salvar arquivos e opções de estilo.
8. **Depuração**: Busque dicas sobre como depurar problemas comuns relacionados a eventos e estilos.
9. **Recursos Adicionais**: Pesquise recursos adicionais para aprofundamento em tópicos específicos, como documentação oficial ou tutoriais avançados.
10. **Exercícios**: Tente aplicar exercícios práticos para cada seção, buscando experimentar e modificar os exemplos fornecidos.

Lembre-se de que a prática é fundamental para dominar esses conceitos mais avançados do Tkinter.

11. Boas Práticas e Dicas: Escrevendo código Tkinter de alta qualidade

Ao trabalhar com Tkinter, especialmente em projetos maiores, a clareza e a organização do código são essenciais. Esta seção destaca as melhores práticas para escrever código Tkinter mais manutenível, reutilizável e menos propenso a erros.

11.1 Organização do Código: Mantendo a ordem em projetos Tkinter

Um código Tkinter bem estruturado é crucial para evitar a "bagunça" que pode surgir ao combinar a lógica da interface com o código da aplicação. Aqui estão algumas práticas recomendadas para manter seu projeto organizado:

1. Separação de Interface e Lógica:

- **Use classes para representar elementos da interface:** Crie classes que herdam de widgets Tkinter (como `tk.Tk`, `ttk.Frame`, `ttk.Button`) para encapsular a aparência e o comportamento de partes específicas da sua interface. Isso torna seu código mais modular e fácil de entender.

```
class TaskEntry(ttk.Frame):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        # ... widgets para entrada de tarefas, prioridade, etc ...

class TaskList(tk.Listbox):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        # ... configurações da Listbox para exibir tarefas ...
```

- **Mantenha a lógica da aplicação fora dos callbacks de eventos:** Em vez de colocar grandes blocos de código dentro de funções como `command` de um botão,

crie métodos separados em suas classes de interface. Isso melhora a legibilidade e facilita a reutilização da lógica em outros lugares.

```
class ToDoApp(tk.Tk):
    # ...

    def add_task(self):
        task = self.task_entry.get()
        # ... lógica para adicionar tarefa ...
        self.task_list.insert(tk.END, task)

# ...

add_button = ttk.Button(..., command=app.add_task)
```

2. Utilize Constantes para Melhorar a Manutenção:

- **Defina cores, fontes e dimensões como constantes:** Isso torna seu código mais fácil de ler e modificar no futuro. Em vez de usar valores literais espalhados pelo código:
- `WINDOW_WIDTH = 400`
- `WINDOW_HEIGHT = 300`
- `PRIMARY_COLOR = "#2196F3"`
  `FONT_FAMILY = "Helvetica"`

- **Centralize as configurações visuais:** Facilita a aplicação de um tema consistente à sua aplicação.

3. Documente seu Código:

- **Use comentários para explicar a lógica:** Não subestime o poder de um comentário bem colocado! Explique o propósito de classes, métodos e seções complexas de código.
- **Docstrings para documentação reutilizável:** Use docstrings para descrever o que suas classes e funções fazem. Isso permite que ferramentas como o `help()` e geradores de documentação forneçam informações úteis.

```
def format_task_string(task, priority, due_date):
    """Formata a string da tarefa para exibição na lista.

    Args:
        task (str): A descrição da tarefa.
        priority (str): A prioridade da tarefa (Baixa, Média, Alta).
        due_date (datetime): A data de vencimento da tarefa.

    Returns:
        str: A string formatada da tarefa.
    """
# ... lógica de formatação ...
```

11.2 Reutilização de Componentes: Criando blocos de construção Tkinter

A reutilização de código é um princípio fundamental em desenvolvimento de software. Em Tkinter, isso significa criar widgets e funcionalidades reutilizáveis para evitar a repetição e simplificar a manutenção.

1. Crie Widgets Personalizados Através de Herança:

- **Especialize o comportamento de widgets:** Se você precisar de um widget com funcionalidade adicional, crie uma nova classe que herde de um widget Tkinter existente.

```
class LabeledEntry(ttk.Frame):
    """Um widget que combina um Label e um Entry."""
    def __init__(self, master, label_text, **kwargs):
        super().__init__(master, **kwargs)
        self.label = ttk.Label(self, text=label_text)
        self.label.pack(side=tk.LEFT)
        self.entry = ttk.Entry(self)
        self.entry.pack(side=tk.LEFT)

    def get(self):
        """Retorna o texto digitado no Entry."""
        return self.entry.get()

    def set(self, value):
        """Define o texto do Entry."""
        self.entry.delete(0, tk.END)
        self.entry.insert(0, value)
```

2. Crie Funções para Reutilizar Configurações e Criação de Widgets:

- **Evite a repetição de código:** Se você estiver criando widgets semelhantes várias vezes, defina funções para centralizar a criação e configuração.

```
def create_labeled_entry(parent, label_text):
    """Cria e retorna um LabeledEntry."""
    frame = ttk.Frame(parent)
    ttk.Label(frame, text=label_text).pack(side=tk.LEFT)
    entry = ttk.Entry(frame)
    entry.pack(side=tk.LEFT)
    return frame, entry

# Uso
task_frame, task_entry = create_labeled_entry(root, "Tarefa:")
task_frame.pack()
```

11.3 Depuração: Desvendando os mistérios da sua interface Tkinter

Depurar aplicações gráficas como as criadas com Tkinter pode ser desafiador. Use estas estratégias para tornar o processo mais eficaz:

1. Imprima, Imprima, Imprima!

- **Use `print()` para verificar valores:** Exiba os valores de variáveis, o estado de widgets e mensagens em pontos estratégicos do código para entender o fluxo de execução.
- **Crie mensagens de depuração informativas:** Em vez de apenas imprimir "chegou aqui", forneça contexto sobre o que está sendo executado.

2. Utilize o Tratamento de Exceções a seu Favor:

- **Capture exceções para evitar falhas abruptas:** Utilize blocos `try...except` para capturar erros e exibir mensagens mais amigáveis ao usuário, além de registrar informações úteis para depuração.

```
try:
    index = self.task_list.curselection()[0]
    # ...
except IndexError:
    messagebox.showwarning("Nenhuma Tarefa Selecionada", "Por favor, selecione uma tarefa na
lista.")
```

3. Ferramentas de Depuração:

- **Depurador do IDE:** A maioria das IDEs Python (como VS Code, PyCharm) possui depuradores integrados. Use-os para acompanhar passo a passo a execução do seu código Tkinter, inspecionar variáveis e entender o comportamento da interface.

Lembre-se: Ao depurar código Tkinter, é útil ter uma compreensão clara de como o loop de eventos funciona. Se o seu código não está se comportando como esperado, certifique-se de que você não está bloqueando o loop principal da interface.

12. Construindo uma To-Do List com Tkinter: Do planejamento à persistência

Nesta seção, vamos aplicar o que aprendemos sobre Tkinter para construir um aplicativo de lista de tarefas funcional, desde o planejamento da interface até a adição de funcionalidades como persistência de dados.

12.1 Projetando a Interface: Um esboço para o sucesso

Antes de escrever qualquer código, é fundamental ter uma visão clara da interface que queremos criar.

1. Identificando os Elementos Essenciais:

- **Campo de Entrada de Tarefas:** Permite que o usuário digite a descrição da nova tarefa.
- **Botão "Adicionar":** Adiciona a tarefa digitada pelo usuário à lista de tarefas.
- **Lista de Tarefas:** Exibe as tarefas adicionadas, permitindo ao usuário visualizar, selecionar e interagir com elas.
- **Botões de Ação:** Oferecem funcionalidades como marcar tarefas como concluídas e remover tarefas da lista.

2. Esboçando o Layout:

Um esboço visual, mesmo simples, ajuda a visualizar a organização dos elementos.

```
+-----------------------------------+
|  Nova Tarefa: [_____________] [+] |
|                                   |
|  Lista de Tarefas:                |
|  [ ] Tarefa 1                     |
|  [x] Tarefa 2 (Concluída)         |
|  [ ] Tarefa 3                     |
+-----------------------------------+
|  [Concluir] [Remover]             |
+-----------------------------------+
```

3. Escolhendo o Gerenciador de Geometria:

Para este projeto, o `grid()` será uma boa escolha, pois nos permite organizar os widgets em linhas e colunas, proporcionando flexibilidade para o layout.

12.2 Implementando a Interface: Dando vida ao projeto

Com o plano em mente, vamos traduzir nosso esboço em código Tkinter:

```
import tkinter as tk
from tkinter import ttk

class ToDoApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("To-Do List")
        self.geometry("400x300")
        self.create_widgets()

    def create_widgets(self):
        # Frame principal para organizar os widgets
        main_frame = ttk.Frame(self, padding="10")
        main_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
        self.columnconfigure(0, weight=1)
        self.rowconfigure(0, weight=1)

        # Campo de entrada e botão "Adicionar"
        self.task_entry = ttk.Entry(main_frame, width=40)
        self.task_entry.grid(row=0, column=0, padx=(0, 5))

        add_button = ttk.Button(main_frame, text="Adicionar", command=self.add_task)
        add_button.grid(row=0, column=1)

        # Lista de tarefas
        self.task_list = tk.Listbox(main_frame, height=10, width=50)
        self.task_list.grid(row=1, column=0, columnspan=2, pady=10)

        # Botões de ação
        complete_button = ttk.Button(main_frame, text="Concluir", command=self.complete_task)
        complete_button.grid(row=2, column=0, sticky=tk.E, padx=(0, 5))

        remove_button = ttk.Button(main_frame, text="Remover", command=self.remove_task)
        remove_button.grid(row=2, column=1, sticky=tk.W)

    def add_task(self):
        # Lógica para adicionar tarefas (ver seção 12.3)
        pass

    def complete_task(self):
        # Lógica para marcar como concluída (ver seção 12.3)
        pass

    def remove_task(self):
        # Lógica para remover tarefas (ver seção 12.3)
        pass

if __name__ == "__main__":
    app = ToDoApp()
    app.mainloop()
```

Neste código:

- Criamos a janela principal (`ToDoApp`) e configuramos seu título e tamanho.
- Utilizamos um `ttk.Frame` para organizar os widgets internos.
- Criamos o campo de entrada (`ttk.Entry`), o botão "Adicionar" (`ttk.Button`) e a lista de tarefas (`tk.Listbox`).
- Adicionamos botões para as ações "Concluir" e "Remover".
- O `grid()` é usado para posicionar os widgets em um layout de grade.

12.3 Adicionando Funcionalidades: Fazendo a lista funcionar

Agora, vamos adicionar as funcionalidades básicas para tornar nossa lista de tarefas interativa:

```
# ... (código da seção 12.2) ...

    def add_task(self):
        task = self.task_entry.get()
        if task:
            self.task_list.insert(tk.END, task)
            self.task_entry.delete(0, tk.END)

    def complete_task(self):
        try:
            index = self.task_list.curselection()[0]
            self.task_list.itemconfig(index, fg="gray")  # Indica tarefa concluída
        except IndexError:
            pass  # Nenhuma tarefa selecionada

    def remove_task(self):
        try:
            index = self.task_list.curselection()[0]
            self.task_list.delete(index)
        except IndexError:
            pass  # Nenhuma tarefa selecionada
```

Implementamos:

- `add_task()`: Adiciona a tarefa digitada à lista.
- `complete_task()`: Marca a tarefa selecionada como concluída, mudando a cor do texto para cinza.
- `remove_task()`: Remove a tarefa selecionada da lista.

12.4 Persistência de Dados: Salvando suas tarefas

Para tornar nosso aplicativo mais útil, vamos adicionar a capacidade de salvar as tarefas em um arquivo e carregá-las quando o aplicativo for iniciado novamente. Usaremos o formato JSON para armazenar os dados.

```python
import tkinter as tk
from tkinter import ttk
import json

class ToDoApp(tk.Tk):
    # ... (código das seções 12.2 e 12.3) ...

    def save_tasks(self):
        tasks = self.task_list.get(0, tk.END)
        with open("tasks.json", "w") as f:
            json.dump(tasks, f)

    def load_tasks(self):
        try:
            with open("tasks.json", "r") as f:
                tasks = json.load(f)
                for task in tasks:
                    self.task_list.insert(tk.END, task)
        except FileNotFoundError:
            pass  # Ignora se o arquivo não existir

    def __init__(self):
        # ... (código do construtor da seção 12.2) ...
        self.load_tasks()  # Carrega tarefas ao iniciar

    # ...

# No final do código, fora da classe:
if __name__ == "__main__":
    app = ToDoApp()
    app.protocol("WM_DELETE_WINDOW", app.save_tasks)  # Salva ao fechar
    app.mainloop()
```

Adicionamos:

- `save_tasks()`: Salva as tarefas em um arquivo JSON.
- `load_tasks()`: Carrega tarefas do arquivo JSON ao iniciar.
- Integramos `load_tasks()` ao construtor e `save_tasks()` ao fechamento da janela.

Com essas funcionalidades, nossa lista de tarefas se torna muito mais útil, permitindo que o usuário salve seu progresso e continue de onde parou.

13. Expandindo seus Horizontes Tkinter: Recursos e Próximos Passos

Parabéns por construir sua própria To-Do List com Tkinter! A jornada no mundo das interfaces gráficas com Python está apenas começando. Esta seção o guiará para recursos valiosos e inspirará você com ideias para projetos futuros.

13.1 A Fonte do Conhecimento: Explorando a Documentação Oficial

A documentação oficial do Tkinter é sua companheira indispensável nesta jornada. Lá você encontrará:

- **Descrição detalhada de todos os widgets:** Explore as opções de personalização, métodos e funcionamento de cada widget.
- **Informações sobre o gerenciamento de geometria:** Domine os layouts com `pack`, `grid` e `place`.
- **Tratamento de eventos e muito mais!**

Link: https://docs.python.org/3/library/tkinter.html

13.2 Expandindo os Limites: Bibliotecas e Ferramentas Indispensáveis

Tkinter é uma base sólida, e você pode enriquecê-la com bibliotecas e extensões poderosas:

ttk (Themed Tkinter): Widgets com aparência moderna e personalizável.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
style = ttk.Style()
style.theme_use('clam') # Ou outro tema disponível
# ... seus widgets ttk ...
root.mainloop()
```

1. **Pillow (PIL Fork):** Manipulação avançada de imagens para interfaces mais ricas.

```
from PIL import Image, ImageTk

image = Image.open("sua_imagem.jpg")
photo = ImageTk.PhotoImage(image)

label = tk.Label(root, image=photo)
label.pack()
```

2. **TkCalendar:** Integração de calendários para seleção de datas.

```python
from tkcalendar import Calendar
cal = Calendar(root, selectmode='day', year=2024, month=9, day=6)
cal.pack()
```

13.3 Despertando a Criatividade: Ideias para Projetos Tkinter

1. **Editor de Texto Aprimorado:** Crie um editor com abas, destaque de sintaxe, busca e substituição avançada.
2. **Jogos Clássicos:** Desenvolva jogos como Jogo da Velha, Snake ou Tetris para aprimorar suas habilidades.
3. **Gerenciador de Tarefas Completo:** Expanda sua To-Do List com recursos como:
    - Prioridades
    - Data de Vencimento
    - Categorização de tarefas
    - Notificações
    - Interface gráfica mais elaborada.
4. **Visualizador de Dados:** Crie dashboards interativos com gráficos e tabelas usando bibliotecas como Matplotlib ou Seaborn.
5. **Ferramentas de Automação:** Crie interfaces gráficas para seus scripts Python, tornando-os mais acessíveis.

Lembre-se: A prática leva à maestria. Explore a documentação, experimente diferentes bibliotecas e, o mais importante, divirta-se criando!

Códigos TO-DO List

```python
import tkinter as tk
from tkinter import ttk, messagebox

class TodoApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Lista de Tarefas")

        # Lista para armazenar as tarefas
        self.tasks = []

        # Criar e configurar widgets
        self.create_widgets()
```

```
    def create_widgets(self):
        # Frame principal
        main_frame = ttk.Frame(self.root, padding="10")
        main_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

        # Entrada de tarefa
        self.task_entry = ttk.Entry(main_frame, width=40)
        self.task_entry.grid(row=0, column=0, padx=5, pady=5)

        # Botão de adicionar tarefa
        add_button = ttk.Button(main_frame, text="Adicionar", command=self.add_task)
        add_button.grid(row=0, column=1, padx=5, pady=5)

        # Lista de tarefas
        self.task_listbox = tk.Listbox(main_frame, width=50, height=10)
        self.task_listbox.grid(row=1, column=0, columnspan=2, padx=5, pady=5)

        # Barra de rolagem
        scrollbar = ttk.Scrollbar(main_frame, orient=tk.VERTICAL, command=self.task_listbox.yview)
        scrollbar.grid(row=1, column=2, sticky=(tk.N, tk.S))
        self.task_listbox.config(yscrollcommand=scrollbar.set)

        # Botão de remover tarefa
        remove_button = ttk.Button(main_frame, text="Remover", command=self.remove_task)
        remove_button.grid(row=2, column=0, padx=5, pady=5)

        # Botão de limpar lista
        clear_button = ttk.Button(main_frame, text="Limpar Lista", command=self.clear_list)
        clear_button.grid(row=2, column=1, padx=5, pady=5)

    def add_task(self):
        task = self.task_entry.get()
        if task:
            self.tasks.append(task)
            self.task_listbox.insert(tk.END, task)
            self.task_entry.delete(0, tk.END)
        else:
            messagebox.showwarning("Aviso", "Por favor, insira uma tarefa.")

    def remove_task(self):
        try:
            index = self.task_listbox.curselection()[0]
            self.task_listbox.delete(index)
            del self.tasks[index]
        except IndexError:
            messagebox.showwarning("Aviso", "Por favor, selecione uma tarefa para remover.")

    def clear_list(self):
        self.task_listbox.delete(0, tk.END)
        self.tasks.clear()

if __name__ == "__main__":
    root = tk.Tk()
    app = TodoApp(root)
    root.mainloop()
```

Código completo

```
import tkinter as tk
from tkinter import ttk
```

```python
import json

class ToDoApp(tk.Tk):
    def __init__(self):
        """Inicializa a aplicação To-Do List."""
        super().__init__()
        self.title("To-Do List")
        self.geometry("400x300")
        self.create_widgets()
        self.load_tasks()  # Carrega tarefas salvas ao iniciar

    def create_widgets(self):
        """Cria os widgets da interface."""

        # Frame principal para organizar os widgets
        main_frame = ttk.Frame(self, padding="10")
        main_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))
        self.columnconfigure(0, weight=1)
        self.rowconfigure(0, weight=1)

        # Campo de entrada para novas tarefas
        self.task_entry = ttk.Entry(main_frame, width=40)
        self.task_entry.grid(row=0, column=0, padx=(0, 5))

        # Botão "Adicionar"
        add_button = ttk.Button(main_frame, text="Adicionar", command=self.add_task)
        add_button.grid(row=0, column=1)

        # Lista de tarefas
        self.task_list = tk.Listbox(main_frame, height=10, width=50)
        self.task_list.grid(row=1, column=0, columnspan=2, pady=10)

        # Botão "Concluir"
        complete_button = ttk.Button(main_frame, text="Concluir", command=self.complete_task)
        complete_button.grid(row=2, column=0, sticky=tk.E, padx=(0, 5))

        # Botão "Remover"
        remove_button = ttk.Button(main_frame, text="Remover", command=self.remove_task)
        remove_button.grid(row=2, column=1, sticky=tk.W)

    def add_task(self):
        """Adiciona uma nova tarefa à lista."""
        task = self.task_entry.get()
        if task:
            self.task_list.insert(tk.END, task)
            self.task_entry.delete(0, tk.END)

    def complete_task(self):
        """Marca a tarefa selecionada como concluída."""
        try:
            index = self.task_list.curselection()[0]
            self.task_list.itemconfig(index, fg="gray")
        except IndexError:
            pass  # Nenhuma tarefa selecionada

    def remove_task(self):
        """Remove a tarefa selecionada da lista."""
        try:
            index = self.task_list.curselection()[0]
            self.task_list.delete(index)
        except IndexError:
            pass  # Nenhuma tarefa selecionada

    def save_tasks(self):
        """Salva as tarefas em um arquivo JSON."""
        tasks = self.task_list.get(0, tk.END)
```

```
        with open("tasks.json", "w") as f:
            json.dump(tasks, f)

    def load_tasks(self):
        """Carrega tarefas de um arquivo JSON."""
        try:
            with open("tasks.json", "r") as f:
                tasks = json.load(f)
                for task in tasks:
                    self.task_list.insert(tk.END, task)
        except FileNotFoundError:
            pass  # Ignora se o arquivo não existir

if __name__ == "__main__":
    app = ToDoApp()
    app.protocol("WM_DELETE_WINDOW", app.save_tasks)  # Salva ao fechar
    app.mainloop()
```

Este código define uma aplicação simples de lista de tarefas com as seguintes funcionalidades:

- **Adicionar Tarefas:** Digite a tarefa no campo de entrada e clique em "Adicionar" ou pressione Enter.
- **Concluir Tarefas:** Selecione uma tarefa na lista e clique em "Concluir". A tarefa ficará cinza para indicar que foi concluída.
- **Remover Tarefas:** Selecione uma tarefa na lista e clique em "Remover".
- **Persistência de Dados:** As tarefas são salvas automaticamente em um arquivo tasks.json ao fechar a aplicação e carregadas ao iniciar.

Versão anterior ... Interface do Aplicativo de Lista de Tarefas

```
import tkinter as tk
from tkinter import ttk
from tkcalendar import DateEntry
import datetime

class ToDoApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("To-Do List")
        self.geometry("400x500")
        self.create_widgets()

    def create_widgets(self):
        # Frame para entrada de nova tarefa
        entry_frame = ttk.Frame(self, padding="10 10 10 0")
        entry_frame.grid(row=0, column=0, sticky=(tk.W, tk.E))

        ttk.Label(entry_frame, text="Nova Tarefa:").grid(row=0, column=0, sticky=tk.W)
        self.task_entry = ttk.Entry(entry_frame, width=30)
        self.task_entry.grid(row=0, column=1, padx=5)
        ttk.Button(entry_frame, text="+", width=3, command=self.add_task).grid(row=0, column=2)

        ttk.Label(entry_frame, text="Prioridade:").grid(row=1, column=0, sticky=tk.W)
        self.priority_var = tk.StringVar(value="Média")
```

```
        priority_combo = ttk.Combobox(entry_frame, textvariable=self.priority_var, values=["Baixa",
"Média", "Alta"])
        priority_combo.grid(row=1, column=1, padx=5, sticky=tk.W)

        ttk.Label(entry_frame, text="Data:").grid(row=2, column=0, sticky=tk.W)
        self.date_entry = DateEntry(entry_frame, width=12, background='darkblue', foreground='white',
borderwidth=2)
        self.date_entry.grid(row=2, column=1, padx=5, sticky=tk.W)

        # Lista de tarefas
        self.task_list = ttk.Treeview(self, columns=("priority", "date"), show="tree headings")
        self.task_list.heading("priority", text="Prioridade")
        self.task_list.heading("date", text="Data")
        self.task_list.grid(row=1, column=0, sticky=(tk.W, tk.E, tk.N, tk.S), padx=10, pady=10)

        # Scrollbar para a lista de tarefas
        scrollbar = ttk.Scrollbar(self, orient=tk.VERTICAL, command=self.task_list.yview)
        scrollbar.grid(row=1, column=1, sticky=(tk.N, tk.S))
        self.task_list.configure(yscrollcommand=scrollbar.set)

        # Botões de ação
        action_frame = ttk.Frame(self, padding="10")
        action
```

continua ...

Alguns links adicionais que podem ser úteis:

https://www.youtube.com/watch?v=vNEwbfsZ-Js

https://www.cursou.com.br/informatica/tkinter-python/#google_vignette

https://pt.slideshare.net/slideshow/python-tkinter-gui-part-1ppt/252279360

https://www.cos.ufrj.br/~bfgoldstein/python/compII/slides/comp2-tkinter-parte2.pdf

https://www.cos.ufrj.br/~bfgoldstein/python/tutorialtkinter.pdf