Instructor: Mohil Mokaria

BOOTCAMP: PYTHON FOR GUI APPLICATIONS
4. GUI Development with Tkinter

Introduction to GUI Programming

Graphical User Interface (GUI) programming is essential for creating applications that require user interaction. It is especially important in scenarios where a visually intuitive interface is necessary, such as in desktop applications, games, and software tools. The primary goal is to create an environment where users can interact with the program using graphical elements like buttons, menus, and windows.

Advantages of GUI Programming

- **User-Friendly:** GUIs provide a more user-friendly experience, allowing users to interact with the software without having to remember and type complex commands.
- **Visual Representation:** Information can be presented graphically, making it easier for users to understand and interpret data.
- **Interactivity:** GUIs allow for interactive elements like buttons, sliders, and checkboxes, enabling users to actively engage with the application.

Principles of UI Design

- **Consistency:** Maintain a consistent design throughout the application. This includes using the same color scheme, fonts, and layout across different screens.
- **Clarity:** Ensure that the interface is clear and easy to understand. Avoid clutter and use descriptive labels for buttons and menu items.
- **Feedback:** Provide feedback to users about the outcome of their actions. For example, display a message when a form is submitted successfully.
- **Simplicity:** Keep the interface simple and avoid unnecessary complexity. A straightforward design reduces the learning curve for users.

Tools of UI Design

- **Sketching:** Begin the design process by sketching the layout and key components on paper. This helps in visualizing the overall structure of the interface.
- **Prototyping:** Use prototyping tools like Figma, Adobe XD, or Sketch to create interactive mockups of the application. Prototypes allow for testing the usability of the design before actual development.

Introduction to Tkinter

Tkinter is the standard GUI (Graphical User Interface) toolkit that comes with Python. It is based on the Tk GUI toolkit and provides a set of tools for creating graphical user interfaces. Tkinter is widely used due to its simplicity, ease of use, and integration with Python. Tkinter stands for "Tk Interface."

Tkinter originated from the Tk GUI toolkit developed by John Ousterhout in the late 1980s. The name Tkinter comes from the "Tk interface" for Python. Tkinter has undergone several updates and improvements over the years, making it a reliable choice for developing GUI applications in Python.

Why use TKinter?

- **Integration with Python:** Tkinter is included with Python, making it easily accessible for Python developers without the need for additional installations.
- **Simplicity and Ease of Use:** Tkinter provides a straightforward way to create GUI applications, making it an excellent choice for beginners. The syntax is clean and easy to understand.
- **Cross-Platform:** Tkinter is cross-platform, meaning applications developed with Tkinter can run on different operating systems without modification.
- **Active Community and Resources:** As Tkinter has been around for a long time, there are extensive resources and a supportive community, making it easier to find solutions to common issues.

Installation

Tkinter usually comes pre-installed with Python. To check if Tkinter is installed on your system, you can open a Python shell and type:

```python
import tkinter as tk
print(tk.TkVersion)
```

If Tkinter is installed, this will print the version number; otherwise, you may need to install it, which can usually be done using a package manager or directly from the Python website.

Creating your first Tkinter Window

```python
import tkinter as tk

# Create the main window
root = tk.Tk()

# Set window title
root.title("Hello, Tkinter!")

# Run the main loop
root.mainloop()
```

Code Breakdown:

- First we import the Tkinter module and rename it as **'tk'** for convenience.
- Then we create the main window for the application. The **'Tk()'** function is used to instantiate the main window. **'root'** is a common variable name for the main window.
- Then we set the title for the window using the **'title()'** function.

Understanding the mainloop

The mainloop is a crucial component in Tkinter applications. It continuously listens for events (like button clicks or key presses) and keeps the program running until the user closes the window.

Tkinter Widgets

1. **Label:** Displays text or images

```
import tkinter as tk

root = tk.Tk()
label = tk.Label(root, text="Hello, Tkinter!")
label.pack()

root.mainloop()
```

2. **Button:** A Clickable button

```
import tkinter as tk

def on_button_click():
    print("Button clicked!")

root = tk.Tk()
button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack()

root.mainloop()
```

3. **Entry:** A single-line text entry field

```
import tkinter as tk

def on_entry_change(event):
    print(entry.get())

root = tk.Tk()
entry = tk.Entry(root)
entry.pack()
entry.bind("<KeyRelease>", on_entry_change)

root.mainloop()
```

4. **Text:** A multiline text entry and display widget

```
import tkinter as tk

root = tk.Tk()
text_widget = tk.Text(root, height=5, width=30)
text_widget.pack()

root.mainloop()
```

5. **Frame:** A container for other widgets

```
import tkinter as tk

root = tk.Tk()
frame = tk.Frame(root, bg="lightblue", width=200, height=100)
frame.pack()

root.mainloop()
```

6. **Canvas:** A drawing area for shape and images

```
import tkinter as tk

root = tk.Tk()
canvas = tk.Canvas(root, width=200, height=100)
canvas.pack()
canvas.create_rectangle(25, 25, 175, 75, fill="green")

root.mainloop()
```

7. **Checkbutton:** A checkbox

```
import tkinter as tk

root = tk.Tk()
var = tk.BooleanVar()
check_button = tk.Checkbutton(root, text="Check me", variable=var)
check_button.pack()

root.mainloop()
```

8. **Radiobutton:** A radio button

```
import tkinter as tk

root = tk.Tk()
var = tk.IntVar()
radio_button1 = tk.Radiobutton(root, text="Option 1", variable=var,
value=1)
radio_button2 = tk.Radiobutton(root, text="Option 2", variable=var,
value=2)
radio_button1.pack()
radio_button2.pack()

root.mainloop()
```

9. **Scale:** A slider control

```
import tkinter as tk

def on_scale_change(value):
    print(value)

root = tk.Tk()
scale = tk.Scale(root, from_=0, to=100, orient=tk.HORIZONTAL,
command=on_scale_change)
scale.pack()

root.mainloop()
```

10. **Lisbox:** A list of selectable items

```
import tkinter as tk

root = tk.Tk()
listbox = tk.Listbox(root)
for item in ["Item 1", "Item 2", "Item 3"]:
    listbox.insert(tk.END, item)
listbox.pack()

root.mainloop()
```

11. **Menu:** A menu bar or popup menu

```python
import tkinter as tk

def file_new():
    print("File > New")

root = tk.Tk()

menu_bar = tk.Menu(root)

file_menu = tk.Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New", command=file_new)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=root.destroy)

menu_bar.add_cascade(label="File", menu=file_menu)

root.config(menu=menu_bar)
root.mainloop()
```

12. **Scrollbar:** A scrollbar for widgets that support scrolling

```python
import tkinter as tk

root = tk.Tk()

text_widget = tk.Text(root, height=5, width=30)
scrollbar = tk.Scrollbar(root, command=text_widget.yview)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
text_widget.config(yscrollcommand=scrollbar.set)

text_widget.pack()

root.mainloop()
```

13. **Spinbox:** A widget for selecting from a range of values

```python
import tkinter as tk

def on_spinbox_change():
    value = spinbox.get()
    print(f"Selected value: {value}")

root = tk.Tk()

spinbox = tk.Spinbox(root, from_=0, to=10, command=on_spinbox_change)
spinbox.pack()

root.mainloop()
```

14. **LabelFrame:** A labeled container for other widgets

```python
import tkinter as tk

root = tk.Tk()

label_frame = tk.LabelFrame(root, text="My Frame")
label_frame.pack()

label = tk.Label(label_frame, text="This is a labeled frame.")
label.pack()

root.mainloop()
```

15. **Message:** Displays multiple lines of text.

```python
import tkinter as tk

root = tk.Tk()

message = tk.Message(root, text="This is a multi-line message. It can
contain a longer text.")
message.pack()

root.mainloop()
```

Configuring and Styling Widgets

Configuring and styling widgets in Tkinter involves using a variety of options and methods to customize their appearance and behavior.

Label

- **text:** Sets the text displayed by the label.
- **font:** Specifies the font used for the text (e.g., font=("Arial", 12)).
- **fg or foreground:** Sets the text color.
- **bg or background:** Sets the background color.

```
label = tk.Label(root, text="Hello, Tkinter!", font=("Arial", 12),
fg="blue", bg="lightgray")
```

Button

- **text:** Sets the text displayed on the button.
- **font:** Specifies the font used for the text on the button.
- **fg or foreground:** Sets the text color.
- **bg or background:** Sets the background color.
- **command:** Specifies the function to be called when the button is clicked.

```
button = tk.Button(root, text="Click Me", command=on_button_click,
bg="green", fg="white")
```

Entry

- **width:** Specifies the width of the entry widget.
- **font:** Specifies the font used for the text in the entry.
- **bg or background:** Sets the background color.

```
entry = tk.Entry(root, width=20, font=("Helvetica", 10),
bg="lightyellow")
```

Text

- **height and width:** Sets the height and width of the text widget.
- **wrap:** Specifies how lines should be wrapped ("none," "char," or "word").
- **font:** Specifies the font used for the text.
- **state:** Sets the state of the widget (tk.NORMAL or tk.DISABLED).

```
text_widget = tk.Text(root, height=5, width=30, wrap=tk.WORD)
text_widget.config(font=("Courier New", 12), state=tk.DISABLED)
```

Frame

- **bg or background:** Sets the background color.
- **bd or borderwidth:** Sets the width of the border around the frame.
- **relief:** Specifies the type of border ("flat," "sunken," "raised," etc.).

```
frame = tk.Frame(root, bg="lightblue", bd=5, relief=tk.RIDGE)
```

Canvas

- **width and height:** Sets the dimensions of the canvas.
- **bg or background:** Sets the background color.

```
canvas = tk.Canvas(root, width=200, height=100, bg="white")
```

Each widget in Tkinter comes with its own set of options that you can use to customize its appearance and behavior. If you want to explore all the available options for each widget, the Tkinter documentation is a great resource. It provides detailed information on how to configure and style widgets to suit your needs.

Geometry/Layout Management

Geometry management in Tkinter refers to the process of organizing and arranging widgets within a window or a container. Tkinter provides three main geometry managers: pack, grid, and place. Each manager has its own approach to organizing widgets within a parent container.

'pack' Geometry Manager

The pack geometry manager organizes widgets in blocks before placing them in the parent widget. It's a simple and convenient way to stack widgets vertically or horizontally.

Syntax:

```
widget.pack(options)
```

Common Options:

- **side:** Specifies the side of the parent widget to which the widget should be packed (tk.TOP, tk.BOTTOM, tk.LEFT, tk.RIGHT).
- **fill:** Determines how the widget expands to fill the available space (tk.NONE, tk.X, tk.Y, tk.BOTH).
- **expand:** A boolean value that indicates whether the widget should expand to fill any extra space.

Example:

```
import tkinter as tk

root = tk.Tk()

label1 = tk.Label(root, text="Label 1")
label1.pack(side=tk.LEFT)

label2 = tk.Label(root, text="Label 2")
label2.pack(side=tk.RIGHT)

root.mainloop()
```

'grid' Geometry Manager

The grid geometry manager organizes widgets in a table-like structure, allowing you to specify the row and column in which each widget should be placed.

Syntax:

```
widget.grid(options)
```

Common Options:

- **row, column:** Specify the row and column index where the widget should be placed.
- **sticky:** Determines where the widget would be placed in the grid cell (tk.W, tk.E, tk.N, tk.S, tk.W+E, tk.W+N, etc.).
- **rowspan, columnspan:** Specifies the number of rows or columns a widget should span.

Example:

```
import tkinter as tk

root = tk.Tk()

label1 = tk.Label(root, text="Label 1")
label1.grid(row=0, column=0)

label2 = tk.Label(root, text="Label 2")
label2.grid(row=0, column=1)

root.mainloop()
```

‘place’ Geometry Manager

The place geometry manager allows you to specify an exact position for a widget within its parent, using either absolute coordinates or relative positioning.

Syntax:

```
widget.place(options)
```

Common Options:

- **x, y:** Specify the x and y coordinates of the widget.
- **relx, rely:** Specify the relative x and y coordinates (a value between 0.0 and 1.0).
- **anchor:** Specifies a reference point for placing the widget (tk.NW, tk.CENTER, tk.SE, etc.).

Example:

```
import tkinter as tk

root = tk.Tk()

label1 = tk.Label(root, text="Label 1")
label1.place(x=20, y=30)

label2 = tk.Label(root, text="Label 2")
label2.place(relx=0.5, rely=0.5, anchor=tk.CENTER)

root.mainloop()
```

Event Handling

Event handling in Tkinter refers to the process of capturing and responding to user-generated events, such as mouse clicks, keyboard inputs, or window resize actions. Tkinter provides a way to bind functions (event handlers) to specific events, allowing your application to respond dynamically to user interactions.

Binding Events:

The bind method is used to associate an event with a callback function. This method is applied to a widget and takes two arguments: the event to bind and the callback function to execute when the event occurs.

Syntax:

```
widget.bind(event, callback_function)
```

Event Callback Function

The callback function is the Python function that gets executed when the associated event occurs. It typically takes one argument, which is the event object containing information about the event (e.g., the mouse position, key pressed, etc.).

```
def callback_function(event):
    # Your code here
    pass
```

List of Events:

1. Mouse Events:

    1.1. **<Button-1>:** Left mouse button click
    1.2. **<Button-2>:** Middle mouse button click
    1.3. **<Button-3>:** Right mouse button click
    1.4. **<B1-Motion>:** Left mouse button drag
    1.5. **<B2-Motion>:** Middle mouse button drag
    1.6. **<B3-Motion>:** Right mouse button drag
    1.7. **<Enter>:** Mouse enters widget
    1.8. **<Leave>:** Mouse leaves widget
    1.9. **<Motion>:** Mouse motion

```python
import tkinter as tk

def on_mouse_click(event):
    print(f"Mouse Clicked at x={event.x}, y={event.y}")

root = tk.Tk()

canvas = tk.Canvas(root, width=300, height=200, bg="white")
canvas.pack()

canvas.bind("<Button-1>", on_mouse_click)

root.mainloop()
```

2. Keyboard Events:

2.1. **<KeyPress>:** Any key press

2.2. **<KeyRelease>:** Any key release

2.3. **<Key>:** Specific key press (e.g., <Key-Return>, <Key-A>)

2.4. **<Control-KeyPress>:** Control key press

2.5. **<Shift-KeyPress>:** Shift key press

2.6. **<Alt-KeyPress>:** Alt key press

```python
import tkinter as tk

def on_key_press(event):
    print(f"Key Pressed: {event.keysym}")

root = tk.Tk()

entry = tk.Entry(root)
entry.pack()

entry.bind("<KeyPress>", on_key_press)

root.mainloop()
```

3. Focus Events:

    3.1. **<FocusIn>:** Widget gains focus

    3.2. **<FocusOut>:** Widget loses focus

```
import tkinter as tk

def on_focus_in(event):
    print("Focus In")

def on_focus_out(event):
    print("Focus Out")

root = tk.Tk()

entry = tk.Entry(root)
entry.pack()

entry.bind("<FocusIn>", on_focus_in)
entry.bind("<FocusOut>", on_focus_out)

root.mainloop()
```

4. Window Events:

    4.1. **<Configure>:** Window configuration change (e.g., resize)

    4.2. **<Map>:** Window is mapped (displayed)

    4.3. **<Unmap>:** Window is unmapped (hidden)

    4.4. **<Destroy>:** Widget is destroyed (closed)

```
import tkinter as tk

def on_configure(event):
                   print(f"Window    Resized:    width={event.width},
height={event.height}")

root = tk.Tk()

root.bind("<Configure>", on_configure)

root.mainloop()
```

5. Miscellaneous Events:

5.1. **<ButtonPress>:** Any button press

5.2. **<ButtonRelease>:** Any button release

```python
import tkinter as tk

def on_button_press(event):
    root.config(bg="lightblue")

def on_button_release(event):
    root.config(bg="white")

root = tk.Tk()
root.title("Button Press Release Example")

label = tk.Label(root, text="Click and hold the left mouse button")
label.pack(pady=20)

root.bind("<ButtonPress-1>", on_button_press)
root.bind("<ButtonRelease-1>", on_button_release)

root.mainloop()
```

Images and Icons in Tkinter

In Tkinter, you can display images and icons using the PhotoImage class for simple images and the PhotoImage class along with the PIL (Python Imaging Library) module for more complex image formats. Additionally, you can set an icon for your Tkinter window using the iconbitmap method.

Displaying Images with 'PhotoImage':

```
import tkinter as tk

root = tk.Tk()
root.title("Image Display Example")

# Create a PhotoImage object for a simple image (GIF, PPM/PGM)
image_path = "path/to/your/image.gif"
photo = tk.PhotoImage(file=image_path)

# Create a Label to display the image
image_label = tk.Label(root, image=photo)
image_label.pack()

root.mainloop()
```

Explanation:

- **tk.PhotoImage(file=image_path):** This line creates a PhotoImage object from the specified image file (image.gif in this case). Note that PhotoImage supports a limited set of image formats, including GIF, PPM, and PGM.
- **tk.Label(root, image=photo):** This line creates a Tkinter Label widget and associates it with the PhotoImage object. The label is then used to display the image.
- **image_label.pack():** This line packs the label into the Tkinter window.

Displaying Images with PIL (for other formats):

Before using PIL you need to install it:

```
pip install pillow
```

```python
import tkinter as tk
from PIL import Image, ImageTk

root = tk.Tk()
root.title("PIL Image Display Example")

# Open an image with PIL
image_path = "path/to/your/image.png"
pil_image = Image.open(image_path)

# Convert the PIL image to a Tkinter PhotoImage
tk_image = ImageTk.PhotoImage(pil_image)

# Create a Label to display the image
image_label = tk.Label(root, image=tk_image)
image_label.pack()

root.mainloop()
```

Explanation:

- **Image.open(image_path):** This line opens the specified image file (image.png) using the Image class from the PIL library.
- **ImageTk.PhotoImage(pil_image):** This line converts the PIL image to a Tkinter-compatible PhotoImage object using the ImageTk module.
- The rest of the code is similar to the previous example, where a Tkinter Label widget is created and packed to display the image.

Setting an Icon for the Tkinter Window:

```python
import tkinter as tk

root = tk.Tk()
root.title("Window Icon Example")

# Set an icon for the Tkinter window
icon_path = "path/to/your/icon.ico"
root.iconbitmap(icon_path)

root.mainloop()
```

Explanation:

- **root.iconbitmap(icon_path):** This line sets the icon for the Tkinter window. The specified icon file (icon.ico) is used to represent the window in the taskbar and the title bar.
- Make sure to replace the placeholder paths ("path/to/your/image.gif", "path/to/your/image.png", and "path/to/your/icon.ico") with the actual paths to your image and icon files.

Threading in Tkinter

Threading in Tkinter can be useful to handle background tasks or long-running processes without freezing the graphical user interface (GUI). Tkinter is not thread-safe, meaning that the GUI should be updated only from the main thread. However, you can use the threading module to perform tasks in the background and update the Tkinter GUI safely.

```python
import tkinter as tk
from threading import Thread
import time

def background_task():
    # Simulate a time-consuming task
    for _ in range(5):
        time.sleep(1)
        print("Working...")

    # Update the label after the task is completed
    root.after(0, update_label)

def start_background_task():
    # Create a thread for the background task
    background_thread = Thread(target=background_task)

    # Start the thread
    background_thread.start()

def update_label():
    label.config(text="Task completed!")

root = tk.Tk()
root.title("Simple Threading Example")

label = tk.Label(root, text="Click the button to start a background task")
label.pack(pady=10)

start_button        =        tk.Button(root,        text="Start        Task",
command=start_background_task)
start_button.pack(pady=10)

root.mainloop()
```

In this example:

- **background_task:** Simulates a time-consuming task (printing "Working..." for 5 seconds) and then calls update_label.
- **start_background_task:** Creates a new thread and starts the background_task in that thread.
- **update_label:** Updates the Tkinter label to indicate that the background task has completed. Instead of directly updating the label from the thread, it uses root.after to schedule the label update in the main thread.

Notes:

- Tkinter is not Thread-Safe: Ensure that Tkinter-related operations are performed in the main thread.
- Use threading.Thread: Create a separate thread for background tasks.
- Avoid Direct GUI Updates from Threads: Use the after method or other thread-safe techniques to update the GUI from the main thread.

Themed Tkinter (TTK)

Themed Tkinter, also known as "ttk" (themed Tkinter), is an extension module for Tkinter that provides access to the Tk themed widget set. Themed Tkinter provides a more modern look and feel to your GUI applications by utilizing the native themes of the operating system.

- **Purpose:** Themed Tkinter (ttk) is an extension module for Tkinter that provides access to a themed widget set. It leverages the native theming capabilities of the operating system to create widgets with a modern and consistent look and feel.
- **Module Import:** To use themed widgets, you need to import ttk from tkinter. For example:

```
from tkinter import ttk.
```

```
      import tkinter as tk
from tkinter import ttk  # Import themed Tkinter

def on_button_click():
    label.config(text="Button Clicked")

# Create a Tkinter window
root = tk.Tk()
root.title("Themed Tkinter Example")

# Create a themed button
themed_button = ttk.Button(root, text="Click Me", command=on_button_click)
themed_button.pack(pady=10)

# Create a themed label
label = ttk.Label(root, text="Welcome to Themed Tkinter")
label.pack(pady=10)

# Run the Tkinter event loop
root.mainloop()
```

- ttk.Button and ttk.Label are used instead of tk.Button and tk.Label to create themed widgets. The appearance is influenced by native system themes of the OS.

Themed Widgets

Themed Tkinter (ttk) introduces a set of themed widgets that provide a modern and consistent look across different operating systems. These widgets have a native appearance, making them visually appealing and ensuring a more seamless integration with the user's desktop environment.

ttk.Button:

- A themed button with native styling.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
themed_button = ttk.Button(root, text="Click Me",
command=on_button_click)
themed_button.pack()
```

ttk.Label:

- A themed label with native styling.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
themed_label = ttk.Label(root, text="Welcome to Themed Tkinter")
themed_label.pack()
```

ttk.Entry:

- A themed entry widget for user input.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
themed_entry = ttk.Entry(root)
themed_entry.pack()
```

ttk.Combobox:

- A themed combobox (drop-down list) for selecting from a list of options.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
themed_combobox = ttk.Combobox(root, values=["Option 1", "Option 2",
"Option 3"])
themed_combobox.pack()
```

ttk.Checkbutton:

- A themed check button for boolean options.

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
themed_checkbutton = ttk.Checkbutton(root, text="Check Me")
themed_checkbutton.pack()
```

These are just a few examples of themed widgets available in ttk. Each of these widgets provides a native look and feel, ensuring a consistent and visually appealing design in your Tkinter applications. You can further customize the appearance of these widgets using styles and themes provided by ttk.

Styling and Customization in ttk using ttk.Style

1. Import ‘ttk.Style’

In order to work with styles in Themed Tkinter, you need to import the ttk.Style class.

```
from tkinter import ttk
```

2. Creating Style Object

Create an instance of the ttk.Style class, which will be used to configure and apply styles to ttk widgets.

```
style = ttk.Style()
```

3. Predefined Styles and Themes

Themed Tkinter comes with predefined themes like "clam", "alt", and "default".

Use the theme_use method to set a specific theme for your application.

```
style.theme_use("clam")
```

4. Configuring Widget Styles

Use the configure method to set styling options for a specific widget class (e.g., "TButton").

```
style.configure("TButton",     padding=(10,     5),     relief="flat",
background="blue", foreground="white")
```

5. Applying Styles to Widgets

When creating a ttk widget, you can specify the desired style using the style option.

```
button = ttk.Button(root, text="Styled Button", style="TButton")
```

6. Accessing theme elements

Use the element_options method to retrieve a tuple of options available for a specific theme element (e.g., "TButton").

```
options = style.element_options("TButton")
```

7. **Creating Custom Styles**

Define your own custom style by providing a unique name and configuring its options.

```
style.configure("Custom.TButton", padding=(10, 5), relief="flat",
background="green", foreground="black")
```

8. **Style Map**

The map method allows you to map style elements to different states (e.g., "active", "disabled").

```
style.map("TButton", background=[("active", "red"), ("disabled",
"gray")])
```

9. **Using 'ttk.Style' for Widgets**

Apply styling directly to a specific ttk widget, like a button.

```
button = ttk.Button(root, text="Styled Button")
button_style = ttk.Style()
button_style.configure("TButton", padding=(10, 5), relief="flat",
background="blue", foreground="white")
button["style"] = "TButton"
```

10. **Resetting Styles**

Use the theme_settings method to reset styles to default values for a specific theme element.

```
style.theme_settings("default", {"TButton": {"padding": (3, 2),
"relief": "flat"}})
```

Example

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

def on_button_click():
    print("Button Clicked")

root = tk.Tk()
root.title("Styling and Customization in ttk")

# Create a themed button with custom style
style = ttk.Style()
style.configure("TButton",       padding=(10,       5),       relief="flat",
background="blue", foreground="white")

themed_button = ttk.Button(root, text="Click Me", command=on_button_click,
style="TButton")
themed_button.pack(pady=10)

root.mainloop()
```