Module 10 - TKINTER

Welcome to Module 10! This module introduces you to **Tkinter**, Python's de-facto standard GUI (Graphical User Interface) toolkit. You'll learn how to create windows, add widgets like labels, buttons, and entry fields, and arrange them using various layout managers.

Chapter 1: Hello World using Tkinter

1.1 What is Tkinter?

- **GUI (Graphical User Interface):** A GUI allows users to interact with software using visual elements like windows, icons, menus, and buttons, instead of text-based commands.
- **Tkinter:** It's a standard Python library used to create desktop GUI applications. It provides a set of tools (widgets) that you can use to build graphical interfaces.
- **Why Tkinter?**
    - **Included with Python:** No extra installation required; it comes with your Python distribution.
    - **Simple to Learn:** Relatively straightforward for beginners to pick up.
    - **Cross-Platform:** Tkinter applications can run on Windows, macOS, and Linux without modification.

Basic Structure of a Tkinter Application:

Every Tkinter application generally follows these steps:

1. **Import Tkinter:** `import tkinter as tk` (common alias).
2. **Create the Root Window:** This is the main window of your application. `root = tk.Tk()`.
3. **Add Widgets:** Create instances of Tkinter widgets (e.g., `Label`, `Button`, `Entry`) and place them inside the root window or other containers.
4. **Configure Widgets (Optional):** Set properties like text, color, size, etc.
5. **Pack/Grid/Place Widgets:** Use a layout manager (`pack()`, `grid()`, or `place()`) to arrange widgets within the window.
6. **Start the Event Loop:** `root.mainloop()`. This continuously listens for events (like mouse clicks, key presses) and updates the GUI. Your application will stay open until the window is closed.

1.2 Your First Tkinter Window: "Hello World!"

Let's create the classic "Hello World!" GUI application.

Python
```
# hello_world_tkinter.py

# Step 1: Import the tkinter module
import tkinter as tk
```

```
# Step 2: Create the main application window (root window)
root = tk.Tk()

# Step 3 (Optional but good practice): Set the window title
root.title("My First Tkinter App")

# Step 4: Create a widget - a Label widget to display text
# tk.Label(parent_widget, text="Your text here")
label = tk.Label(root, text="Hello, Tkinter World!")

# Step 5: Arrange the widget using a layout manager (pack() is simplest for
now)
# pack() places widgets in a block.
label.pack()

# Step 6: Start the Tkinter event loop
# This keeps the window open and responsive to user interactions
root.mainloop()

print("Application closed.") # This line will execute only after the
Tkinter window is closed.
```

What you will see:

A small window will appear with the title "My First Tkinter App" and the text "Hello, Tkinter World!" centred within it.

- **`tk.Tk()`**: Creates the fundamental window for your application.
- **`root.title()`**: Sets the text that appears in the title bar of the window.
- **`tk.Label()`**: A widget used to display static text or images. We create one and pass `root` as its parent, meaning it belongs to the `root` window.
- **`label.pack()`**: This is a **layout manager**. `pack()` is the simplest, telling the label to "pack itself" into the window. It automatically adjusts its size to fit the label.
- **`root.mainloop()`**: This is the heart of any Tkinter application. It starts the event loop, which means the application continuously listens for events (like you clicking the close button). Without `mainloop()`, the window would appear and immediately disappear.

Chapter 2: Basic Modifications using Tkinter

Now that you have a basic window, let's explore how to customize its appearance and the widgets within it.

2.1 Window Properties

You can modify properties of the main window using various methods on the `root` object.

- **`root.title("New Title")`**: Changes the text displayed in the window's title bar. (Already seen)
- **`root.geometry("WxH+X+Y")`**:
  - Sets the size and position of the window.
  - `W`: width, `H`: height (in pixels).
  - `X`: x-coordinate, `Y`: y-coordinate (position from top-left of screen).
  - Example: `"500x300"` (500 pixels wide, 300 pixels high).
  - Example: `"500x300+100+50"` (500x300, starting 100 pixels from left, 50 from top).
- **`root.resizable(width, height)`**:
  - Controls whether the user can resize the window horizontally and/or vertically.
  - `True` (or `1`) allows resizing, `False` (or `0`) prevents it.
  - Example: `root.resizable(False, False)` makes the window fixed size.
- **`root.iconbitmap("path/to/icon.ico")`**:
  - Sets a custom icon for the window (usually a `.ico` file on Windows, other formats on other OS).
  - This path is relative to your script or an absolute path.

2.2 Widget Properties (Labels Revisited)

Widgets also have many configurable properties. We'll explore some for the `Label` widget.

- **`fg` (foreground):** Sets the text color.
- **`bg` (background):** Sets the widget's background color.
  - You can use color names (e.g., "red", "blue", "lightgreen") or hexadecimal codes (e.g., "#FF0000" for red).
- **`font`:** Sets the font family, size, and style.
  - Example: `("Arial", 16, "bold")`
- **`padx`, `pady`:** Internal padding.
  - `padx`: Horizontal padding *inside* the widget, between text/image and the widget's border.
  - `pady`: Vertical padding *inside* the widget.
- **`relief`:** Sets the border style of the widget.
  - Common values: `tk.FLAT`, `tk.RAISED`, `tk.SUNKEN`, `tk.GROOVE`, `tk.RIDGE`.
- **`bd` (border width):** Sets the width of the border (in pixels) for `relief`.
- **`width`, `height`:** Sets the fixed width and height of the widget.
  - For text widgets (like Label), `width` is in text characters, `height` is in text lines.

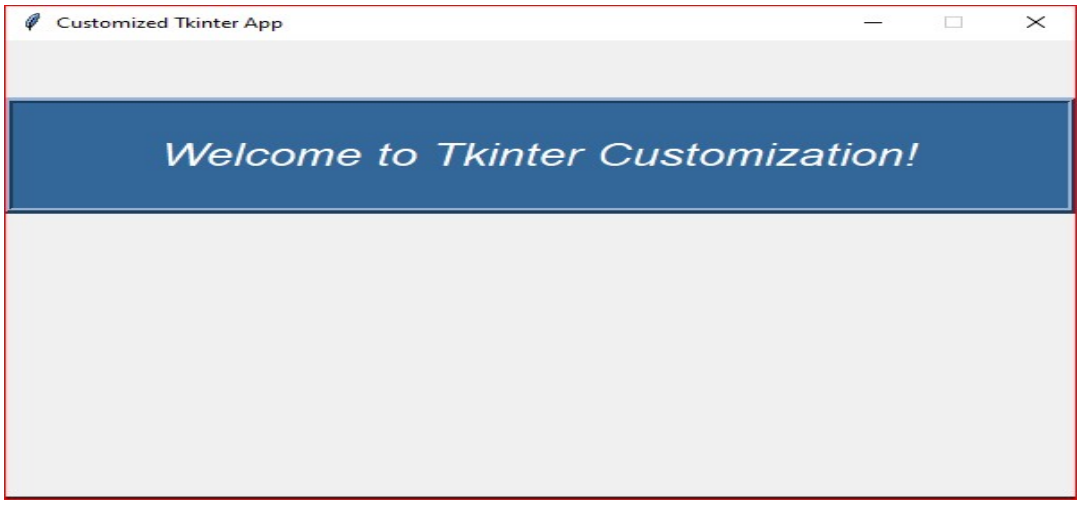
2.3 Examples: Customizing Window and Label

Python

```
# basic_modifications.py
import tkinter as tk

root = tk.Tk()
root.title("Customized Tkinter App")

# Set window size and position (widthxheight+x_pos+y_pos)
root.geometry("600x400+200+100") # 600x400 window, 200px from left, 100px
from top

# Prevent window resizing
root.resizable(False, False)

# Try setting an icon (make sure 'python_icon.ico' exists in the same
directory)
# root.iconbitmap("python_icon.ico")

# Create a customized Label
custom_label = tk.Label(
    root,
    text="Welcome to Tkinter Customization!",
    fg="white",                 # Foreground (text) color
    bg="#336699",               # Background color (a shade of blue)
    font=("Helvetica", 20, "italic"), # Font family, size, style
    padx=20,                    # Horizontal internal padding
    pady=15,                    # Vertical internal padding
    relief=tk.RIDGE,            # Border style
    bd=5,                       # Border width
    width=35,                   # Fixed width in characters
    height=2                    # Fixed height in lines
)

custom_label.pack(pady=50) # Add some external padding from top of window

root.mainloop()
```

What you will see:

A blue-ish window with a ridge border, white italic text, and fixed dimensions. You won't be able to resize it.



Chapter 3: Frames and Buttons using Tkinter

As your applications grow, you'll need to organize widgets. **Frames** are excellent for this. **Buttons** are fundamental for user interaction.

3.1 The `Frame` Widget

- **Purpose:** A `Frame` widget acts as a container for other widgets. It helps in grouping related widgets together and applying layout managers to them independently. This makes your GUI cleaner and easier to manage.
- Think of a `Frame` as a panel or a section within your window.

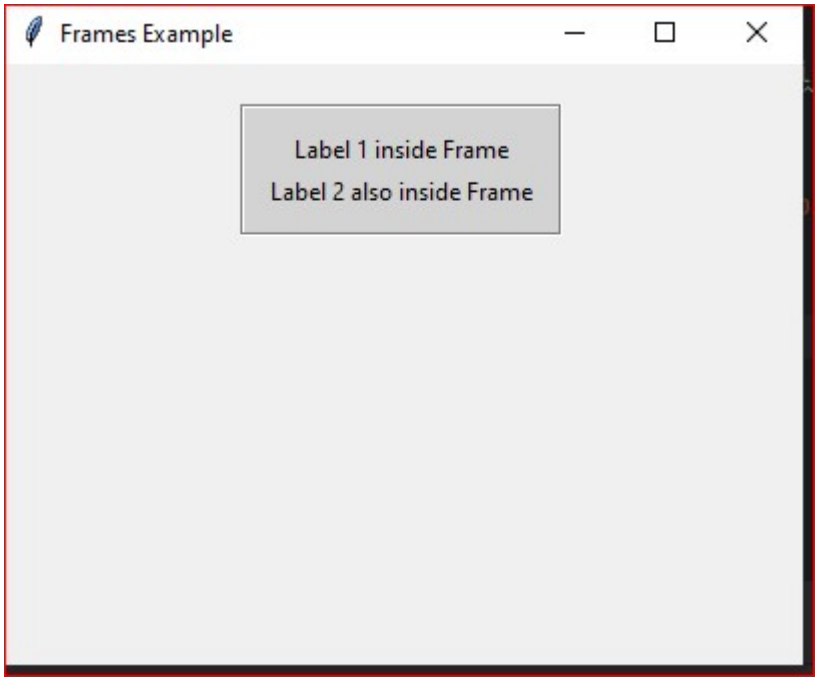
Creating and Packing Frames:

Python

```
# frame_example.py
import tkinter as tk

root = tk.Tk()
root.title("Frames Example")
root.geometry("400x300")

# Create a Frame
# tk.Frame(parent_widget, [options])
my_frame = tk.Frame(root, bd=2, relief=tk.GROOVE, bg="lightgray", padx=10,
pady=10)

# Pack the frame into the root window
my_frame.pack(pady=20) # Add external padding for the frame itself

# Now, add widgets to the frame (NOT directly to root)
label1 = tk.Label(my_frame, text="Label 1 inside Frame", bg="lightgray")
label1.pack()

label2 = tk.Label(my_frame, text="Label 2 also inside Frame",
bg="lightgray")
label2.pack()

root.mainloop()
```

What you will see:

A window with a grey frame in the centre. Inside this frame, two labels will be stacked vertically.



3.2 The `Button` Widget

- **Purpose:** A `Button` widget is used to trigger an action or execute a command when clicked by the user.
- **Key Property: `command`**
  - The `command` option is crucial for buttons. You assign a function (without parentheses `()`) to it. When the button is clicked, that function will be executed.

Creating a Button:

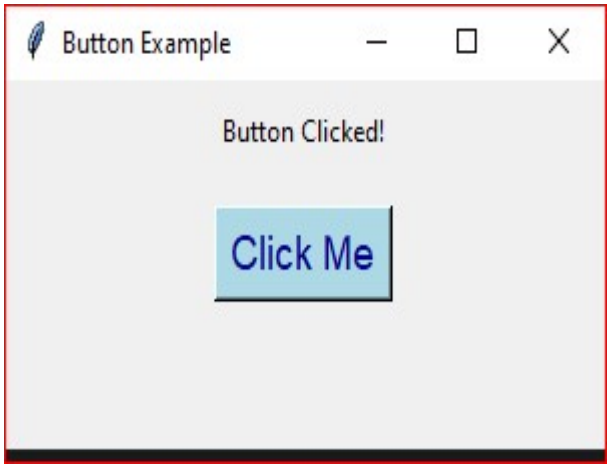
Python

```
# button_example.py
import tkinter as tk

def on_button_click():
    """Function to be called when the button is clicked."""
    print("Button was clicked!")
    # We can also update a label in the GUI
    status_label.config(text="Button Clicked!")

root = tk.Tk()
root.title("Button Example")
root.geometry("300x150")

# Create a Label to show status messages
status_label = tk.Label(root, text="Click the button!")
status_label.pack(pady=10)

# Create a Button
# tk.Button(parent_widget, text="Display Text", command=function_to_call)
my_button = tk.Button(
    root,
    text="Click Me",
    command=on_button_click, # Link the button to the function
```

```
    font=("Arial", 14),
    bg="lightblue",
    fg="darkblue",
    activebackground="darkblue", # Color when button is pressed
    activeforeground="white"
)

my_button.pack(pady=10)

root.mainloop()
```

What you will see:

A window with a label and a button. When you click the "Click Me" button, "Button was clicked!" will print to your console, and the label text in the GUI will change to "Button Clicked!".



Chapter 4: Entry Box and Grid Layout

This chapter introduces how to get user input using an `Entry` widget and a powerful layout manager: `grid()`.

4.1 The `Entry` Widget (Input Field)

- **Purpose:** The `Entry` widget is used for accepting single-line text input from the user. (For multi-line input, you'd use a `Text` widget, which is more advanced).
- **Key Methods:**
  - `entry_widget.get()`: Retrieves the current text entered in the entry box.
  - `entry_widget.insert(index, string)`: Inserts `string` at the specified `index`. `tk.END` inserts at the end.
  - `entry_widget.delete(first_index, last_index)`: Deletes characters from `first_index` up to (but not including) `last_index`. `0, tk.END` deletes all text.

4.2 The `grid()` Layout Manager

- **Purpose:** The `grid()` geometry manager organizes widgets in a table-like structure of rows and columns. It's much more structured and flexible than `pack()` for complex forms.
- **Key Options:**
  - `row`: The row number where the widget will be placed (starting from 0).
  - `column`: The column number where the widget will be placed (starting from 0).
  - `rowspan`: How many rows the widget should span.
  - `columnspan`: How many columns the widget should span.
  - `padx`, `pady`: External padding (space outside the widget, between it and its neighbors/cell borders).
  - `ipadx`, `ipady`: Internal padding (space *inside* the widget, between its content and its border).
  - `sticky`: Specifies how the widget should align within its grid cell if the cell is larger than the widget. Use `tk.N`, `tk.S`, `tk.E`, `tk.W` (North, South, East, West) or combinations (e.g., `tk.NSEW` to make it expand in all directions).

4.3 Examples: Simple Login Form with `Entry` and `grid()`

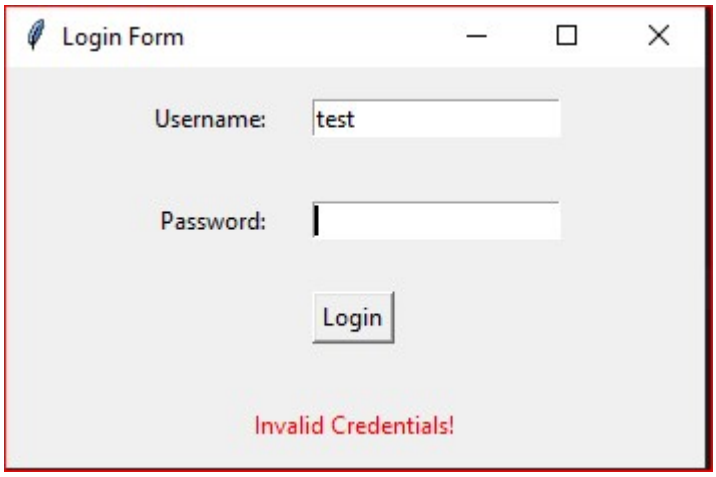
Python

```
# login_form.py
import tkinter as tk

def login_attempt():
    username = username_entry.get()
    password = password_entry.get()
    if username == "admin" and password == "password123":
        result_label.config(text="Login Successful!", fg="green")
    else:
        result_label.config(text="Invalid Credentials!", fg="red")
    # Clear password field after attempt
    password_entry.delete(0, tk.END)

root = tk.Tk()
root.title("Login Form")
root.geometry("350x200") # Set initial window size

# Use grid layout for better alignment

# Label for Username
username_label = tk.Label(root, text="Username:")
# Place at row 0, column 0, align to East
username_label.grid(row=0, column=0, padx=10, pady=10, sticky=tk.E)

# Entry for Username
username_entry = tk.Entry(root)
# Place at row 0, column 1, expand horizontally (W-E)
username_entry.grid(row=0, column=1, padx=10, pady=10, sticky=tk.W)

# Label for Password
password_label = tk.Label(root, text="Password:")
password_label.grid(row=1, column=0, padx=10, pady=10, sticky=tk.E)

# Entry for Password (show asterisks for security)
```

```
password_entry = tk.Entry(root, show="*")
password_entry.grid(row=1, column=1, padx=10, pady=10, sticky=tk.W)

# Login Button
login_button = tk.Button(root, text="Login", command=login_attempt)
# Place at row 2, column 0, spanning 2 columns, center it with sticky=tk.N
login_button.grid(row=2, column=0, columnspan=2, pady=10, sticky=tk.N)

# Result Label
result_label = tk.Label(root, text="")
result_label.grid(row=3, column=0, columnspan=2, pady=5)

# Optional: Make columns/rows expand when window resizes
root.grid_columnconfigure(0, weight=1) # Column 0 expands
root.grid_columnconfigure(1, weight=1) # Column 1 expands
root.grid_rowconfigure(0, weight=1)
root.grid_rowconfigure(1, weight=1)
root.grid_rowconfigure(2, weight=1)
root.grid_rowconfigure(3, weight=1)


root.mainloop()
```

What you will see:

A simple login form with "Username" and "Password" labels, their respective entry boxes, and a "Login" button. Below the button, a label will display "Login Successful!" or "Invalid Credentials!" based on your input.



Chapter 5: Pack Layout Manager (Detailed)

While `grid()` offers precise control, `pack()` is simpler for stacking or arranging widgets side-by-side. Let's revisit `pack()` in more detail.

5.1 Review of `pack()`

- **How it works:** `pack()` places widgets in a block-like fashion. It's often compared to stacking items in a box.
- **Default Behavior:** By default, widgets `pack()` themselves `side=tk.TOP` and are centred horizontally. They try to take up as little space as possible.

5.2 Key `pack()` Options

- **`side`:** Determines which side of the parent widget the current widget will be packed against.
  - `tk.TOP` (default): Packs at the top.
  - `tk.BOTTOM`: Packs at the bottom.
  - `tk.LEFT`: Packs to the left.
  - `tk.RIGHT`: Packs to the right.
- **`fill`:** Specifies whether the widget should expand to fill any extra space in the parent window.
  - `tk.NONE` (default): Widget does not expand.
  - `tk.X`: Expands horizontally (fills available width).
  - `tk.Y`: Expands vertically (fills available height).
  - `tk.BOTH`: Expands both horizontally and vertically.
- **`expand`:**
  - `False` (default): The widget does not take up extra space in the parent.
  - `True`: The widget takes up any extra space in the parent window not used by other widgets. Often used with `fill`.
- **`padx`, `pady` (External Padding):**
  - Adds space *outside* the widget, between it and its neighbors.
  - `padx`: Horizontal padding.
  - `pady`: Vertical padding.
- **`ipadx`, `ipady` (Internal Padding):**
  - Adds space *inside* the widget, between its content and its border.

5.3 Examples: Using `pack()` Effectively

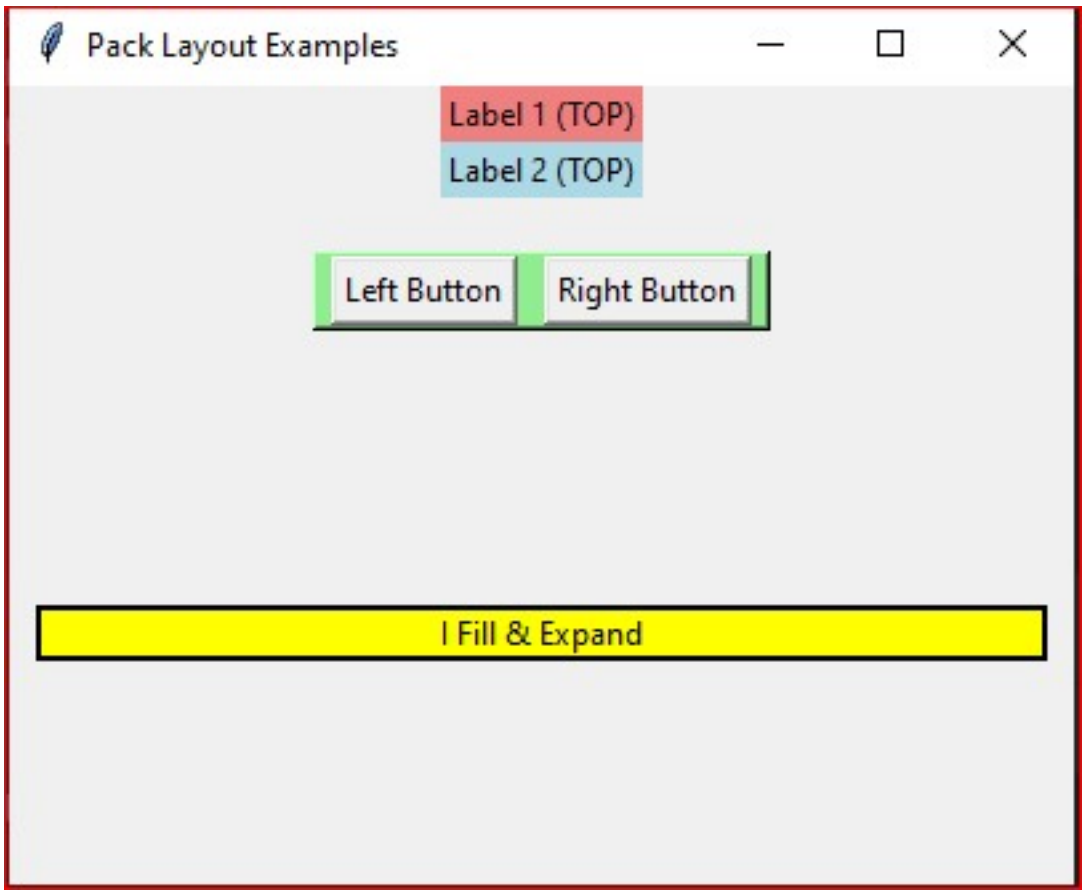
Python
```
# pack_layout_examples.py
import tkinter as tk

root = tk.Tk()
root.title("Pack Layout Examples")
root.geometry("400x300")

# --- Example 1: Basic Stacking (Default side=TOP) ---
label1 = tk.Label(root, text="Label 1 (TOP)", bg="lightcoral")
label1.pack() # Default: side=tk.TOP, fill=tk.NONE, expand=False

label2 = tk.Label(root, text="Label 2 (TOP)", bg="lightblue")
label2.pack() # Default: side=tk.TOP, fill=tk.NONE, expand=False

# --- Example 2: Side-by-Side ---
# Create a frame to contain side-by-side buttons
side_frame = tk.Frame(root, bd=2, relief=tk.RAISED, bg="lightgreen")
side_frame.pack(pady=20) # Pack the frame itself

btn_left = tk.Button(side_frame, text="Left Button")
```

```
btn_left.pack(side=tk.LEFT, padx=5) # Pack to the left of the frame

btn_right = tk.Button(side_frame, text="Right Button")
btn_right.pack(side=tk.RIGHT, padx=5) # Pack to the right of the frame

# --- Example 3: Fill and Expand ---
# Label that fills horizontally and expands
label_fill_expand = tk.Label(root, text="I Fill & Expand", bg="yellow",
bd=2, relief=tk.SOLID)
label_fill_expand.pack(side=tk.BOTTOM, fill=tk.X, expand=True, padx=10,
pady=10)

root.mainloop()
```

What you will see:

1. Two labels stacked at the top (`Label 1`, `Label 2`).
2. Below them, a raised green frame with two buttons: "Left Button" on the left and "Right Button" on the right.
3. At the very bottom, a yellow label that stretches across the width of the window and expands vertically if you resize the window.

- `pack()` is simple and effective for layouts that are primarily linear (stacking or tiling). For more complex, grid-like forms, `grid()` is usually preferred.



Chapter 6: Handling Button Clicks and States

You've already seen the basic `command` option for buttons. Let's explore how to pass arguments to command functions and manage button states.

6.1 Linking Buttons to Functions (Revisit `command` with Arguments)

- The `command` option of a button expects a function reference *without* parentheses `()`. If you put parentheses, the function will execute immediately when the program starts, not when the button is clicked.
- **Problem:** How do you pass arguments to the function then?
- **Solution: `lambda` functions:** Use a `lambda` (anonymous) function to wrap your function call with arguments. The `lambda` itself is the callable object passed to `command`.

  Python

```
# button_args_state.py
import tkinter as tk

def greet(name):
    print(f"Hello, {name}!")
    greeting_label.config(text=f"Hello, {name}!")

def set_button_state(state):
    if state == "disable":
        my_button.config(state=tk.DISABLED)
        print("Button disabled.")
    elif state == "enable":
        my_button.config(state=tk.NORMAL)
        print("Button enabled.")

root = tk.Tk()
root.title("Button Arguments & State")
root.geometry("300x200")

greeting_label = tk.Label(root, text="Waiting for click...")
greeting_label.pack(pady=10)

# Button with no arguments
my_button = tk.Button(root, text="Click Me (No Args)",
command=lambda: greet("World"))
my_button.pack(pady=5)

# Button with argument using lambda
greet_alice_button = tk.Button(root, text="Greet Alice",
command=lambda: greet("Alice"))
greet_alice_button.pack(pady=5)

# Buttons to control main button's state
disable_button = tk.Button(root, text="Disable Button",
command=lambda: set_button_state("disable"))
disable_button.pack(side=tk.LEFT, padx=5)

enable_button = tk.Button(root, text="Enable Button", command=lambda:
set_button_state("enable"))
```

```
enable_button.pack(side=tk.RIGHT, padx=5)

root.mainloop()
```

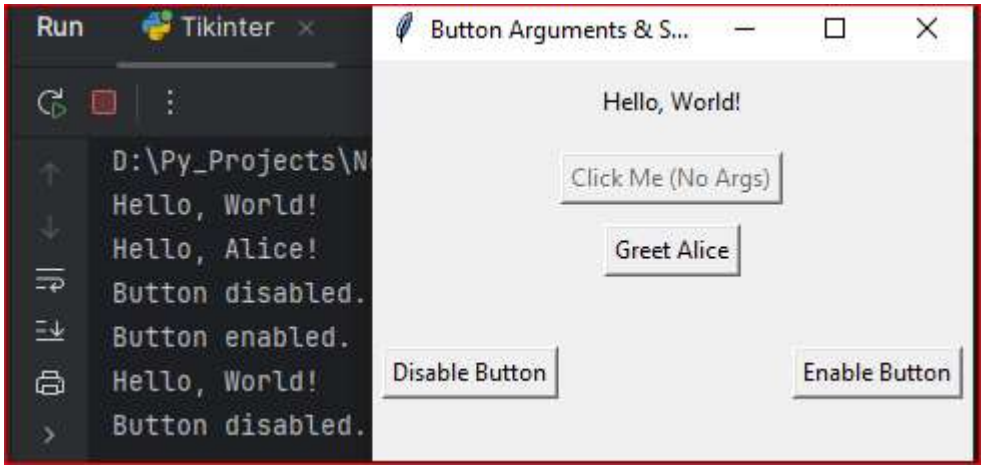
6.2 Button States

- Buttons can be in different states that affect their appearance and responsiveness.
- **`state` property:**
  - `tk.NORMAL` (default): The button is active and can be clicked.
  - `tk.DISABLED`: The button is grayed out and cannot be clicked.
- You can change the button's state dynamically using the `config()` method, as shown in the example above.

What you will see:

A window with a label and four buttons.

- "Click Me (No Args)" and "Greet Alice" will change the label and print to console.
- "Disable Button" will make the "Click Me (No Args)" button unclickable and gray it out.
- "Enable Button" will restore the "Click Me (No Args)" button to its normal state.



Chapter 7: Menubar

Menubars are common in desktop applications, providing a structured way to offer options to the user (e.g., File, Edit, Help).

7.1 Introduction to Menubars

- A `Menu` widget in Tkinter can serve as a top-level menubar for the entire application or as a context (right-click) menu for a specific widget.
- We'll focus on creating a top-level menubar.

Steps to Create a Menubar:

1. Create a `tk.Menu` object, passing the `root` window as its parent.
2. Use `root.config(menu=menu_object)` to attach this menu to the root window as its menubar.
3. Create individual menus (e.g., "File", "Edit") as `tk.Menu` objects, making the main menubar their parent.
4. Add commands (`add_command`) or submenus (`add_cascade`) to these individual menus.
5. Add these individual menus to the main menubar using `add_cascade`.

7.2 Adding Menu Items

- **`menu.add_command(label="Text", command=function_name)`**:
  - Adds a standard, clickable menu item. `label` is the text displayed, `command` is the function to call.
- **`menu.add_separator()`**:
  - Adds a horizontal line to visually separate groups of menu items.
- **`menu.add_cascade(label="SubMenuName", menu=submenu_object)`**:
  - Creates a submenu. `label` is the text displayed for this item in the parent menu, and `menu` is the `tk.Menu` object for the submenu itself.

7.3 Linking Menu Items to Functions

As with buttons, use the `command` option to link menu items to Python functions.

7.4 Example: Simple Editor Menu

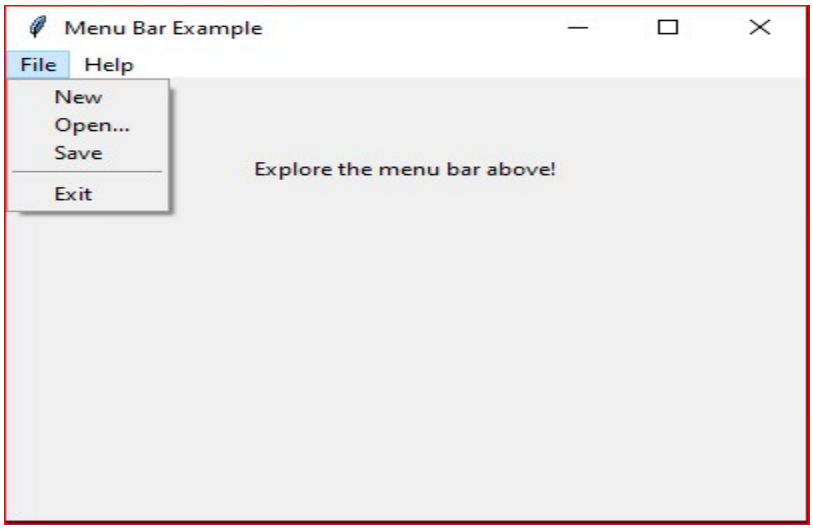
Python

```
# menubar_example.py
import tkinter as tk
from tkinter import messagebox # We'll cover this properly in next chapters

def new_file():
    print("New File created!")
    messagebox.showinfo("File", "Creating a new file...")

def open_file():
    print("Opening File...")
    messagebox.showinfo("File", "Opening an existing file...")

def save_file():
    print("Saving File...")
    messagebox.showinfo("File", "Saving current file...")

def exit_app():
    if messagebox.askyesno("Exit", "Are you sure you want to exit?"):
        root.destroy() # Closes the Tkinter window

def show_about():
    messagebox.showinfo("About", "Simple Menu Bar Example v1.0")

root = tk.Tk()
root.title("Menu Bar Example")
```

```
root.geometry("400x300")

# 1. Create the main menubar
main_menu = tk.Menu(root)
root.config(menu=main_menu) # Attach it to the root window

# 2. Create 'File' menu (a submenu)
file_menu = tk.Menu(main_menu, tearoff=0) # tearoff=0 prevents a dashed
line at top
main_menu.add_cascade(label="File", menu=file_menu) # Add 'File' to main
menubar

# Add commands to 'File' menu
file_menu.add_command(label="New", command=new_file)
file_menu.add_command(label="Open...", command=open_file)
file_menu.add_command(label="Save", command=save_file)
file_menu.add_separator() # Add a separator line
file_menu.add_command(label="Exit", command=exit_app)

# 3. Create 'Help' menu (another submenu)
help_menu = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="Help", menu=help_menu) # Add 'Help' to main
menubar

# Add commands to 'Help' menu
help_menu.add_command(label="About", command=show_about)

# Add a simple label to the main window
info_label = tk.Label(root, text="Explore the menu bar above!")
info_label.pack(pady=50)

root.mainloop()
```

What you will see:

A window with a traditional menu bar at the top (File, Help). Clicking "File" will reveal New, Open, Save, a separator, and Exit. Clicking "Help" will show "About". Selecting these options will print to the console and/or trigger a small pop-up message box (introduced briefly here, covered more later).



Chapter 8: Message Box (`tkinter.messagebox`)

Tkinter's `messagebox` module provides convenient pop-up dialogs to inform users, ask questions, or display warnings/errors.

8.1 Purpose

- To display simple, non-interactive messages to the user.
- To get a simple confirmation (Yes/No, OK/Cancel) from the user (covered in Chapter 10).
- These are modal dialogs, meaning the user must interact with them before returning to the main application.

8.2 Basic Message Box Types (Informational)

These functions display a message and an "OK" button. They primarily differ in their icon and title.

- **`tkinter.messagebox.showinfo(title, message)`**:
  - Displays an informational message with a blue "i" icon.
- **`tkinter.messagebox.showwarning(title, message)`**:
  - Displays a warning message with a yellow exclamation mark icon.
- **`tkinter.messagebox.showerror(title, message)`**:
  - Displays an error message with a red "X" icon.

8.3 Example

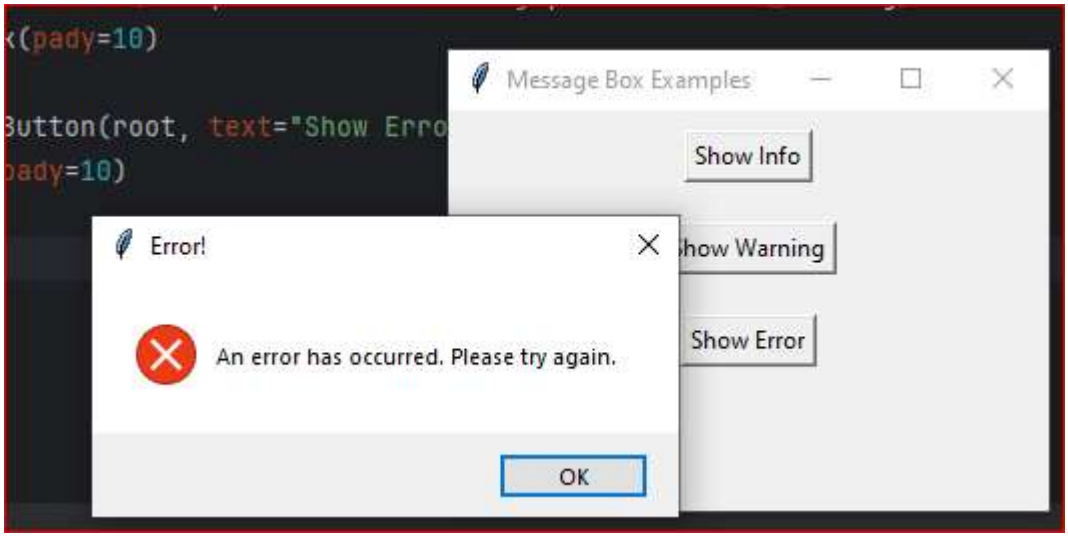
Python

```
# messagebox_basic.py
import tkinter as tk
from tkinter import messagebox # Import the messagebox module

def show_info():
    messagebox.showinfo("Information", "This is an informational message.")

def show_warning():
    messagebox.showwarning("Warning!", "Be careful! This is a warning
message.")

def show_error():
    messagebox.showerror("Error!", "An error has occurred. Please try
again.")

root = tk.Tk()
root.title("Message Box Examples")
root.geometry("300x200")

info_button = tk.Button(root, text="Show Info", command=show_info)
info_button.pack(pady=10)

warning_button = tk.Button(root, text="Show Warning", command=show_warning)
warning_button.pack(pady=10)

error_button = tk.Button(root, text="Show Error", command=show_error)
```

```
error_button.pack(pady=10)

root.mainloop()
```

What you will see:

A window with three buttons. Clicking each button will pop up a different type of message box (information, warning, error), each with its distinct icon and a single "OK" button to dismiss it.



Chapter 9: Drawing on Canvas

The `Canvas` widget allows you to create custom graphics, draw shapes, and even display images within your Tkinter application.

9.1 The `Canvas` Widget

- **Purpose:** A versatile widget used for displaying structured graphics. You can draw lines, rectangles, circles, text, images, and even other widgets on a canvas.
- It's like a blank drawing board within your GUI.

Creating a Canvas:

Python
```
canvas = tk.Canvas(parent_widget, width=W, height=H, bg="color")
```

9.2 Drawing Basic Shapes

The `Canvas` widget provides methods to create various graphical items. All drawing methods return an ID for the created item, which you can use later to modify or delete it.

- **`create_line(x1, y1, x2, y2, ..., options)`**: Draws one or more connected lines.
    - `fill`: Color of the line.
    - `width`: Thickness of the line.
- **`create_rectangle(x1, y1, x2, y2, options)`**: Draws a rectangle.
    - `x1, y1`: Top-left corner coordinates.
    - `x2, y2`: Bottom-right corner coordinates.
    - `fill`: Fill color.
    - `outline`: Border color.
- **`create_oval(x1, y1, x2, y2, options)`**: Draws an oval (ellipse) inscribed within the rectangle defined by `(x1, y1)` and `(x2, y2)`. For a circle, make it a square.
- **`create_polygon(x1, y1, x2, y2, ..., options)`**: Draws a polygon defined by a sequence of coordinates.
- **`create_text(x, y, text="text", options)`**: Draws text on the canvas.
    - `x, y`: Coordinates for the text (default center).
    - `anchor`: Alignment of text relative to `(x,y)` (e.g., `tk.NW` for North-West/top-left).

9.3 Colors and Coordinates

- **Colors:** You can use color names (e.g., "red", "blue") or hex codes (e.g., "#FF0000").
- **Coordinates:** All coordinates are in pixels, with `(0, 0)` being the **top-left corner** of the canvas. X-values increase to the right, Y-values increase downwards.

9.4 Example: Simple Drawing App

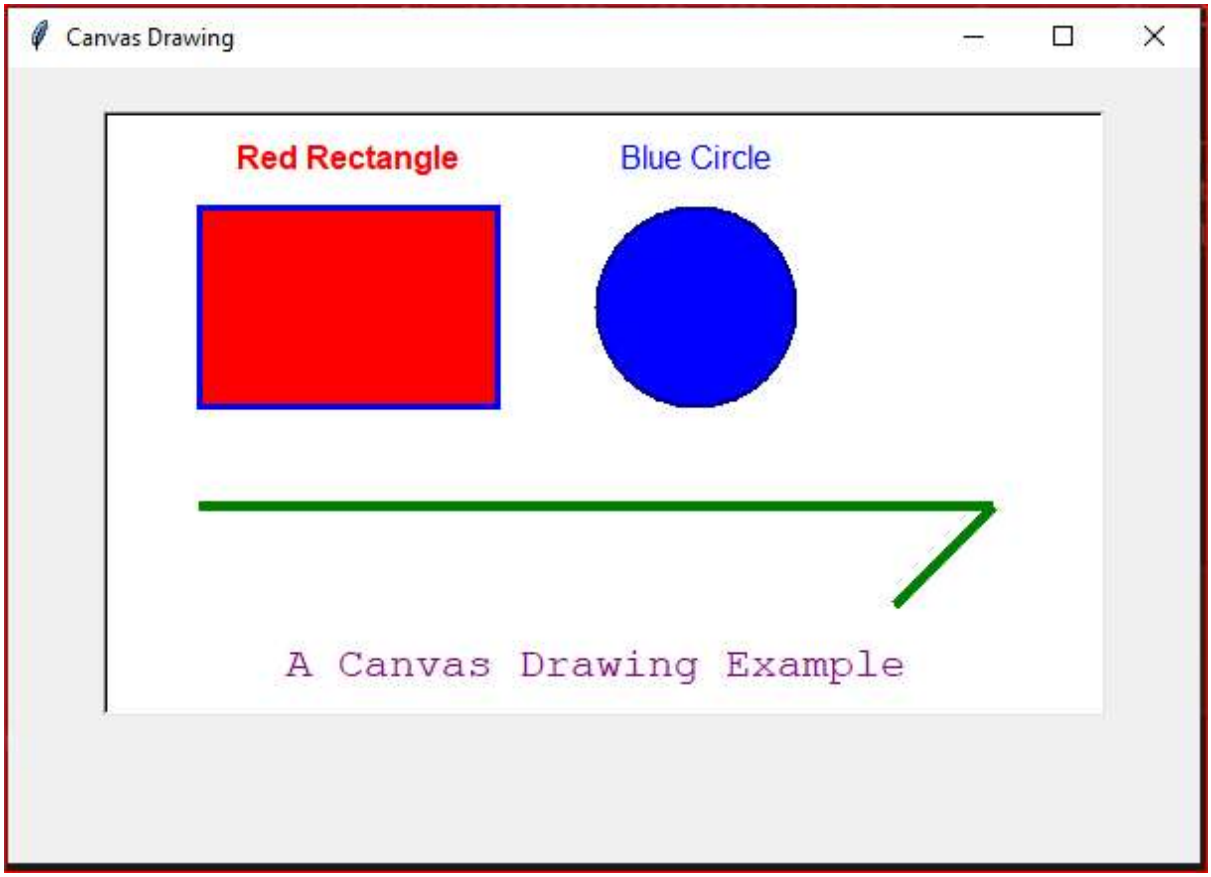
Python

```
# canvas_drawing.py
import tkinter as tk

root = tk.Tk()
root.title("Canvas Drawing")
root.geometry("600x400")

# Create a Canvas widget
canvas = tk.Canvas(root, width=500, height=300, bg="white", bd=2,
relief=tk.SUNKEN)
canvas.pack(pady=20)

# 1. Draw a red rectangle
# (x1, y1) is top-left, (x2, y2) is bottom-right
canvas.create_rectangle(50, 50, 200, 150, fill="red", outline="blue",
width=3)

# 2. Draw a blue circle (oval inscribed in a square)
# (x1, y1, x2, y2) defines the bounding box for the oval
canvas.create_oval(250, 50, 350, 150, fill="blue", outline="darkblue",
width=2)

# 3. Draw a green line
canvas.create_line(50, 200, 450, 200, fill="green", width=5)
canvas.create_line(450, 200, 400, 250, fill="green", width=5) # Connected
line
```

```
# 4. Draw some text
canvas.create_text(125, 25, text="Red Rectangle", fill="red",
font=("Arial", 12, "bold"))
canvas.create_text(300, 25, text="Blue Circle", fill="blue", font=("Arial",
12))
canvas.create_text(250, 280, text="A Canvas Drawing Example",
fill="purple", font=("Courier", 16), anchor=tk.CENTER)

root.mainloop()
```

What you will see:

A window displaying a white canvas with a sunken border. On the canvas, you'll see a red rectangle, a blue circle, a green line, and some descriptive text placed at different positions.



Chapter 10: Message Box (Part 2: Confirmation and Questions)

Beyond simple informational pop-ups, `tkinter.messagebox` can be used to ask the user questions and get specific responses.

10.1 User Input/Confirmation Message Boxes

These functions display a message with multiple buttons (e.g., Yes/No, OK/Cancel) and return a specific value based on the user's choice.

- **`tkinter.messagebox.askquestion(title, message)`**:
  - Displays a message with "Yes" and "No" buttons.
  - Returns the string `"yes"` or `"no"`.
- **`tkinter.messagebox.askokcancel(title, message)`**:
  - Displays a message with "OK" and "Cancel" buttons.
  - Returns `True` if "OK" is clicked, `False` if "Cancel" is clicked.
- **`tkinter.messagebox.askyesno(title, message)`**:
  - Displays a message with "Yes" and "No" buttons.
  - Returns `True` if "Yes" is clicked, `False` if "No" is clicked.
- **`tkinter.messagebox.askretrycancel(title, message)`**:
  - Displays a message with "Retry" and "Cancel" buttons.
  - Returns `True` if "Retry" is clicked, `False` if "Cancel" is clicked.

10.2 Handling User Responses

You need to check the return value of these functions to determine the user's choice and act accordingly.

10.3 Example: Exit Confirmation

Python

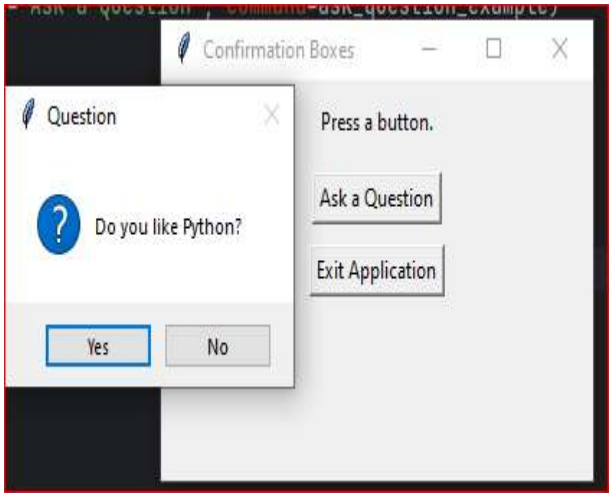
```
# messagebox_confirm.py
import tkinter as tk
from tkinter import messagebox

def confirm_exit():
    # askyesno returns True for Yes, False for No
    response = messagebox.askyesno("Exit Application", "Do you really want
to exit?")
    if response: # If user clicked Yes
        root.destroy() # Close the main window
    else:
        print("Exit cancelled.")
        status_label.config(text="Exit cancelled.")

def ask_question_example():
    response = messagebox.askquestion("Question", "Do you like Python?")
    if response == "yes":
        print("User likes Python!")
        status_label.config(text="User likes Python!")
    else:
        print("User does not like Python.")
        status_label.config(text="User does not like Python.")

root = tk.Tk()
root.title("Confirmation Boxes")
root.geometry("300x200")

status_label = tk.Label(root, text="Press a button.")
status_label.pack(pady=10)
```

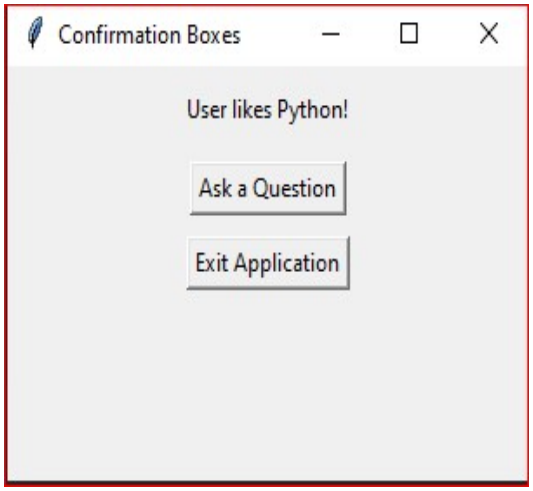
```
question_button = tk.Button(root, text="Ask a Question",
command=ask_question_example)
question_button.pack(pady=5)

exit_button = tk.Button(root, text="Exit Application",
command=confirm_exit)
exit_button.pack(pady=5)

root.mainloop()
```

What you will see:

A window with two buttons.

- Clicking "Ask a Question" will pop up a message box with "Yes" and "No" buttons. Your choice will be printed to the console and displayed in the status label.
- Clicking "Exit Application" will pop up another message box asking for confirmation to exit. If you click "Yes," the application closes; if "No," it remains open.





Chapter 11: Check Box (`Checkbutton`)

The `Checkbutton` widget is used when you want to allow the user to select one or more options from a set. Each check box operates independently.

11.1 Purpose

- To present a binary choice (on/off, true/false) for an option.
- Allows multiple independent selections. (For mutually exclusive choices, use `Radiobutton` which we won't cover in this module but operates similarly).

11.2 Creating a `Checkbutton`

- **`tk.Checkbutton(parent_widget, options)`**:
  - `text`: The label displayed next to the checkbox.

- **`variable`**: This is crucial. You must associate an `IntVar` (for integer values, typically 0 or 1) or `StringVar` (for string values) with the `Checkbutton`. This variable will hold the current state of the checkbox.
- `onvalue`: The value assigned to `variable` when the checkbox is checked (default is 1 for `IntVar`).
- `offvalue`: The value assigned to `variable` when the checkbox is unchecked (default is 0 for `IntVar`).
- `command` (optional): A function to call whenever the checkbox state changes.

11.3 Getting and Setting Checkbox State

- **Getting:** Access the value of the associated `tk.IntVar()` or `tk.StringVar()` using its `.get()` method.
- **Setting:** Set the value of the associated `tk.IntVar()` or `tk.StringVar()` using its `.set()` method.

11.4 Example: Order Options

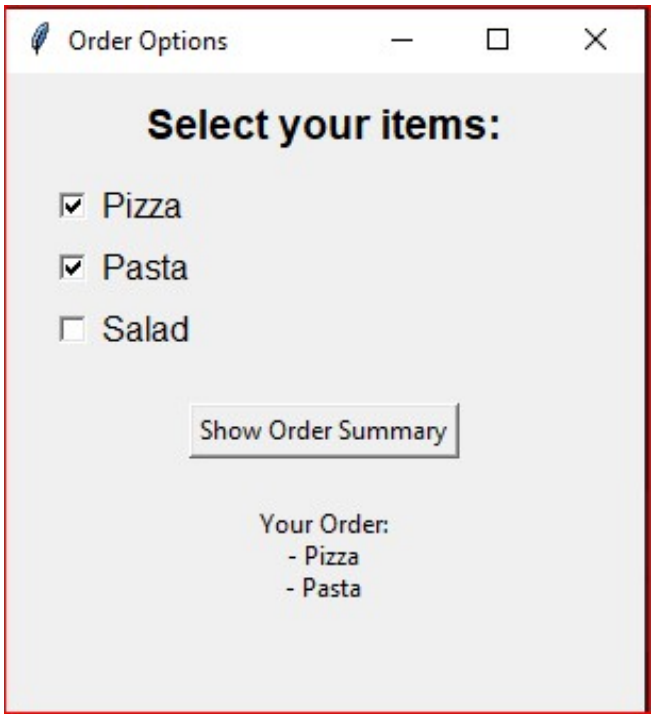
Python

```
# checkbox_example.py
import tkinter as tk

def show_order_summary():
    summary_text = "Your Order:\n"
    # Get the state of each Checkbutton using its associated variable
    if pizza_var.get() == 1: # Check if onvalue (1) is set
        summary_text += "- Pizza\n"
    if pasta_var.get() == 1:
        summary_text += "- Pasta\n"
    if salad_var.get() == 1:
        summary_text += "- Salad\n"

    if summary_text == "Your Order:\n": # If nothing selected
        summary_text = "No items selected."

    summary_label.config(text=summary_text)

root = tk.Tk()
root.title("Order Options")
root.geometry("300x300")

tk.Label(root, text="Select your items:", font=("Arial", 14,
"bold")).pack(pady=10)

# Create IntVar variables to hold the state of each Checkbutton
pizza_var = tk.IntVar()
pasta_var = tk.IntVar()
salad_var = tk.IntVar()

# Create Checkbuttons, associating each with its variable
# onvalue=1, offvalue=0 are defaults for IntVar, explicitly included for
clarity
pizza_cb = tk.Checkbutton(root, text="Pizza", variable=pizza_var,
onvalue=1, offvalue=0, font=("Arial", 12))
pizza_cb.pack(anchor=tk.W, padx=20) # Align to West (left)
```

```
pasta_cb = tk.Checkbutton(root, text="Pasta", variable=pasta_var,
onvalue=1, offvalue=0, font=("Arial", 12))
pasta_cb.pack(anchor=tk.W, padx=20)

salad_cb = tk.Checkbutton(root, text="Salad", variable=salad_var,
onvalue=1, offvalue=0, font=("Arial", 12))
salad_cb.pack(anchor=tk.W, padx=20)

# Button to show selected items
summary_button = tk.Button(root, text="Show Order Summary",
command=show_order_summary)
summary_button.pack(pady=20)

# Label to display the summary
summary_label = tk.Label(root, text="")
summary_label.pack()

root.mainloop()
```

What you will see:

A window with three checkboxes labeled "Pizza", "Pasta", and "Salad". You can check any combination of these. Below them, a button "Show Order Summary". Clicking this button will update a label with the list of selected items.



Chapter 12: Place Layout Manager

The `place()` geometry manager offers the most precise control over widget positioning, allowing you to specify exact coordinates or relative positions.

12.1 Purpose

- **Absolute Positioning:** You can specify exact X and Y coordinates (in pixels) relative to the top-left corner of the parent widget.
- **Relative Positioning:** You can specify positions and sizes as percentages of the parent widget's dimensions.
- **When to Use:** `place()` is useful for very specific, non-resizing layouts, for overlaying widgets, or when designing custom, complex interfaces where `pack()` or `grid()` might be too rigid.

12.2 Key `place()` Options

- **`x`, `y`:** Absolute x and y coordinates (in pixels) of the top-left corner of the widget.
- **`relx`, `rely`:** Relative x and y coordinates (float from 0.0 to 1.0) of the widget's top-left corner, relative to the parent's width/height. `relx=0.5` means 50% across the parent.
- **`width`, `height`:** Absolute width and height of the widget (in pixels).
- **`relwidth`, `relheight`:** Relative width and height (float from 0.0 to 1.0) as a fraction of the parent's width/height.
- **`anchor`:** Specifies which part of the widget is placed at the `(x, y)` or `(relx, rely)` coordinates. Default is `tk.NW` (top-left). Other options include `tk.CENTER`, `tk.N`, `tk.S`, `tk.E`, `tk.W`, `tk.NE`, `tk.SE`, `tk.SW`.

12.3 When to Use `place()`

- **Custom Design:** When you need pixel-perfect control over widget placement for a unique visual layout.
- **Overlapping Widgets:** `place()` allows widgets to overlap, which `pack()` and `grid()` generally prevent.
- **Fixed-Size Windows:** It works best for windows that are not expected to be resized often, as dynamic resizing with `place()` can be complex (unless you use `relx`, `rely`, `relwidth`, `relheight`).

12.4 Comparison with `pack()` and `grid()`

Feature	`pack()`	`grid()`	`place()`
Philosophy	Block-based, flow layout (stacking/tiling)	Table-based (rows/columns)	Absolute or relative positioning
Flexibility	Simple for linear layouts, less for complex	Very flexible for structured forms	Most flexible for custom, non-standard layouts
Resizing	`fill`, `expand` for basic resizing	`rowconfigure`, `columnconfigure` for dynamic resizing	`relx/y/width/height` for relative resizing; hard for absolute

Feature	`pack()`	`grid()`	`place()`
Ease of Use	Easiest for simple UIs	Good balance of control and ease	Can be complex for complex layouts, error-prone for responsiveness
Overlapping	No	No	Yes

12.5 Example: Overlaying Widgets

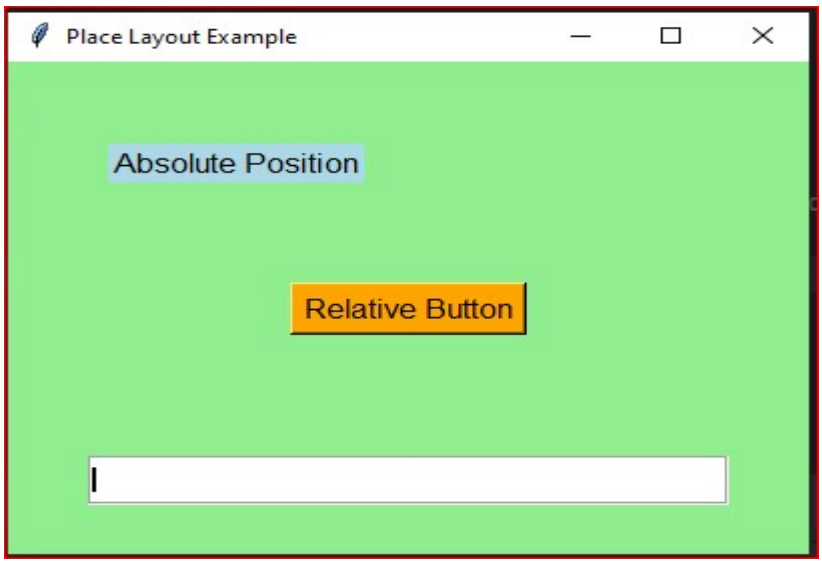
Python

```
# place_layout_example.py
import tkinter as tk

root = tk.Tk()
root.title("Place Layout Example")
root.geometry("400x300")

# Create a background label to fill the window
bg_label = tk.Label(root, bg="lightgreen")
# Use place to make it fill the entire parent (root)
bg_label.place(x=0, y=0, relwidth=1, relheight=1)

# A label placed using absolute coordinates
label_abs = tk.Label(root, text="Absolute Position", bg="lightblue",
font=("Arial", 12))
label_abs.place(x=50, y=50) # 50 pixels from top-left

# A button placed using relative coordinates
button_rel = tk.Button(root, text="Relative Button", bg="orange",
font=("Arial", 12))
# Place its center at 50% width, 50% height of parent
button_rel.place(relx=0.5, rely=0.5, anchor=tk.CENTER)

# An entry box with relative width and absolute height
entry_rel_size = tk.Entry(root, bd=2, relief=tk.GROOVE)
entry_rel_size.place(relx=0.1, rely=0.8, relwidth=0.8, height=30) # 10%
from left, 80% down, 80% width

root.mainloop()
```

What you will see:

A light green window.

- A "Absolute Position" label in the top-left quadrant (fixed position).
- A "Relative Button" centred in the window (it will stay centred if you resize).
- An entry box stretching across 80% of the window's width at the bottom (also resizing horizontally with the window).



Chapter 13: Coding Challenges for Tkinter

Time to build some GUI applications using the concepts you've learned!

Challenge 1: Simple Calculator GUI

Goal: Create a basic calculator interface with an entry box for display and buttons for digits and operations.

Concepts Covered: `Entry`, `Button`, `grid()` layout, `lambda` for button commands, updating `Entry` widget.

Requirements:

1. Create a main window.
2. Use a single `Entry` widget at the top to display input/output. Make it read-only for output.
3. Arrange buttons for digits (0-9) and basic operations (`+`, `-`, `*`, `/`, `=`, `C` for clear) using the `grid()` layout manager.
4. Implement the logic to:
    - Append digits/operators to the entry.
    - Perform calculation when `=` is pressed.
    - Clear the entry when `C` is pressed.
    - (Bonus: Handle basic error cases like division by zero).

Hint: Store the current expression as a string and use Python's `eval()` for calculation (be cautious with `eval()` in real-world apps, but it's fine for this simple challenge).

Challenge 2: To-Do List App

Goal: Build a simple to-do list application where users can add and remove tasks.

Concepts Covered: `Entry`, `Button`, `Label`, `Listbox`, `Frame` (for organization), `pack()` or `grid()`.

Requirements:

1. Create a main window with a title.
2. Have an `Entry` widget for typing new tasks.
3. Have an "Add Task" button. When clicked, it should add the text from the `Entry` to a `Listbox` and clear the `Entry`.
4. Have a "Remove Selected Task" button. When clicked, it should remove the currently selected task from the `Listbox`.
5. Use a `Listbox` to display the tasks.
6. (Optional: Add error handling for empty task input or no selection for removal).

Hint: For `Listbox`:

- `listbox.insert(tk.END, item)` to add.
- `listbox.curselection()` returns a tuple of indices of selected items.
- `listbox.delete(index)` to remove.

Challenge 3: Temperature Converter

Goal: Create an application that converts temperature between Celsius and Fahrenheit.

Concepts Covered: `Entry`, `Label`, `Button`, `grid()` or `pack()`, input validation.

Requirements:

1. Create a window.
2. Have two `Entry` widgets: one for Celsius, one for Fahrenheit.
3. Have labels next to each `Entry` indicating "Celsius" and "Fahrenheit".
4. Have two buttons:
    - "Convert to Fahrenheit": When pressed, takes the value from the Celsius entry, converts it, and displays the result in the Fahrenheit entry.
    - "Convert to Celsius": When pressed, takes the value from the Fahrenheit entry, converts it, and displays the result in the Celsius entry.
5. Implement the conversion formulas:
    - Celsius to Fahrenheit: F=Ctimesfrac95+32
    - Fahrenheit to Celsius: C=(F−32)timesfrac59
6. Add basic error handling: if the user enters non-numeric input, display a message (e.g., using `messagebox.showerror`).