GUI Programming using Tkinter

Getting Started with Python's GUI Toolkit

Welcome to this practical introduction to Tkinter, Python's standard library for creating graphical user interfaces. This presentation will guide you through the fundamentals of building interactive desktop applications with Python's most accessible GUI toolkit.

Getting Started with Tkinter

Tkinter (short for "Tk interface") is Python's standard GUI (Graphical User Interface) package that comes pre-installed with Python. It provides a powerful, platform-independent way to create desktop applications with graphical elements.

To begin using Tkinter, simply import it in your Python script:

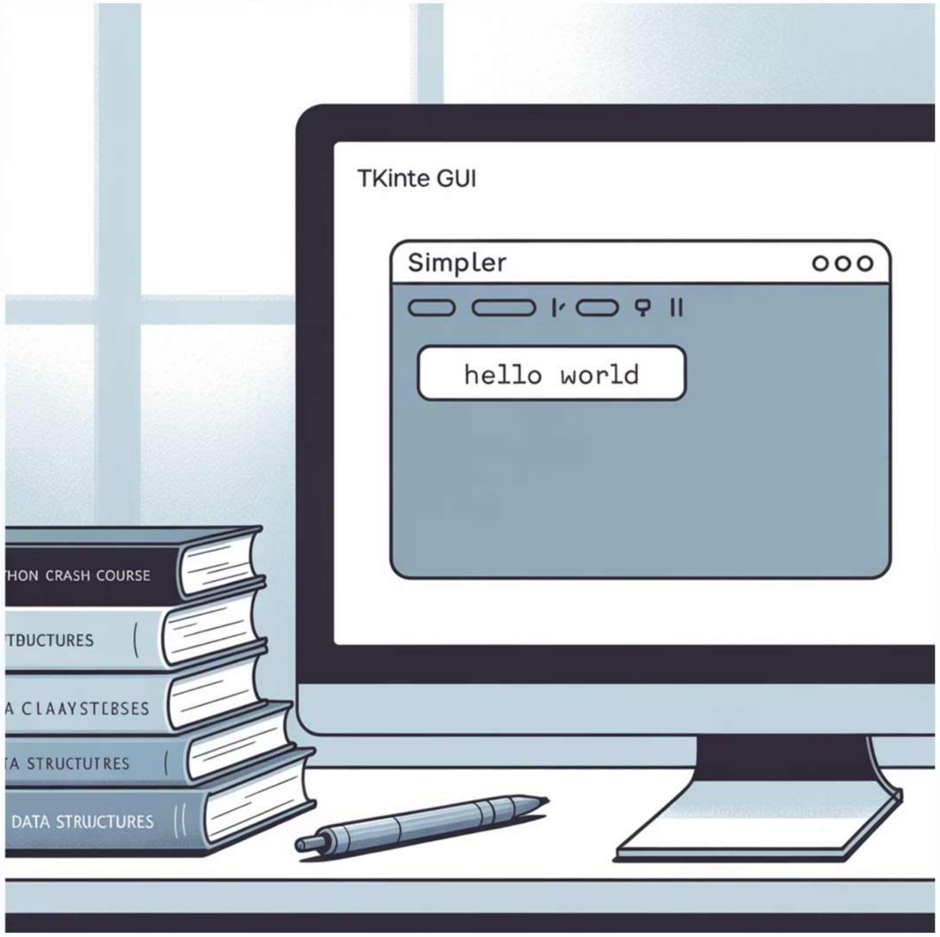
```
import tkinter as tk
# Create the main application window
root = tk.Tk()
root.title("My First Tkinter App")
# Set window size
root.geometry("400x300")
# Start the event loop
root.mainloop()
```

Here, root usually refers to the **main window** created with tk.Tk()

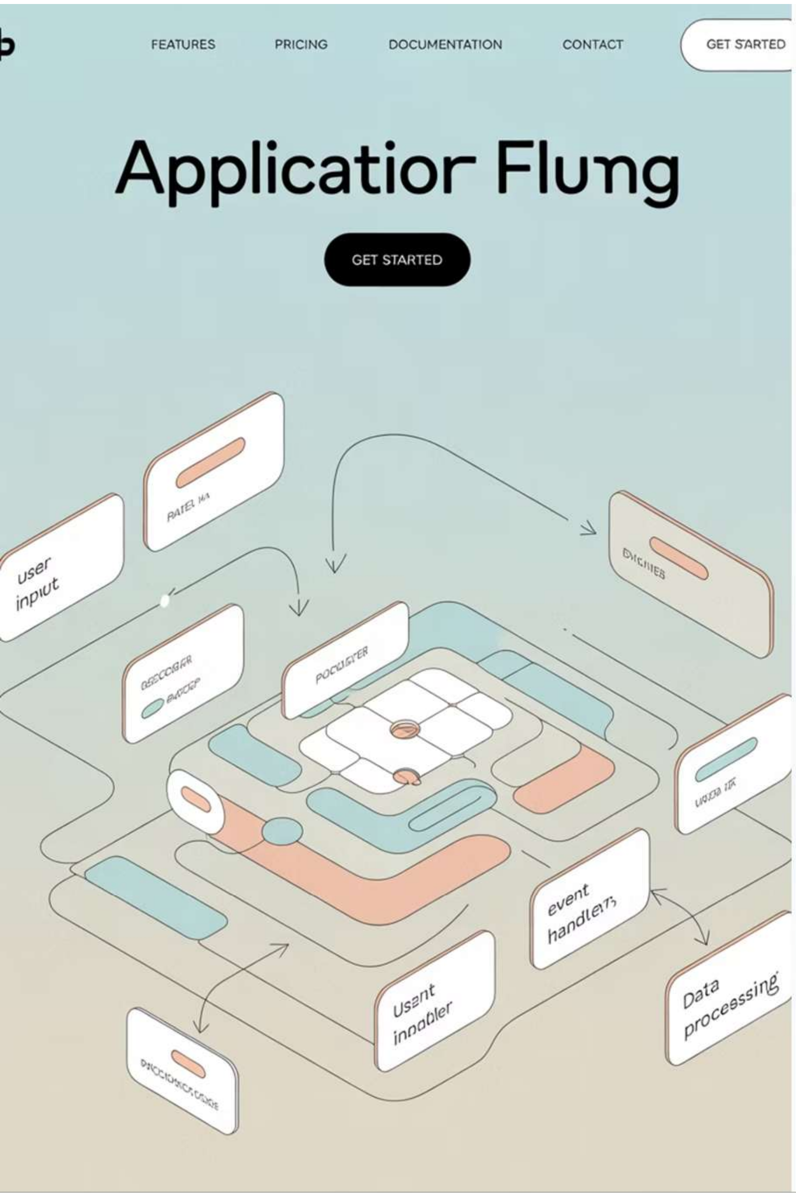
When you create a Tkinter window (root = tk.Tk()), the window is prepared but not yet responsive.
Calling root.mainloop() tells Tkinter:

- **Keep the window open** until the user closes it.
- **Listen for events** (mouse clicks, key presses, button clicks, etc.).
- **Call the appropriate callback functions** when events happen.

Without mainloop(), the window would appear for a split second and then the program would immediately exit.



The `mainloop()` method starts the Tkinter event loop, which waits for and processes user events until the window is closed. This is always the last line in your Tkinter application.



Processing Events

Event-Driven Programming

Tkinter applications follow an event-driven paradigm where the program responds to user actions rather than executing in a predetermined sequence.

```
def button_click():
label.config(text="Button clicked!")

button = tk.Button(root,
text="Click Me",
command=button_click)
```

The Event Loop

The `mainloop()` function creates an infinite loop that:

- Waits for events (mouse clicks,
- key presses)
- Processes them according to event
- handlers
- Updates the display as needed
- Continues until the window is closed

Binding Events

For more complex event handling, use the `bind()` method:

```
def on_key_press(event):
            print(f"Key pressed: {event.char}")

root.bind("", on_key_press)
```

This attaches specific event types to any widget.

Widget Classes

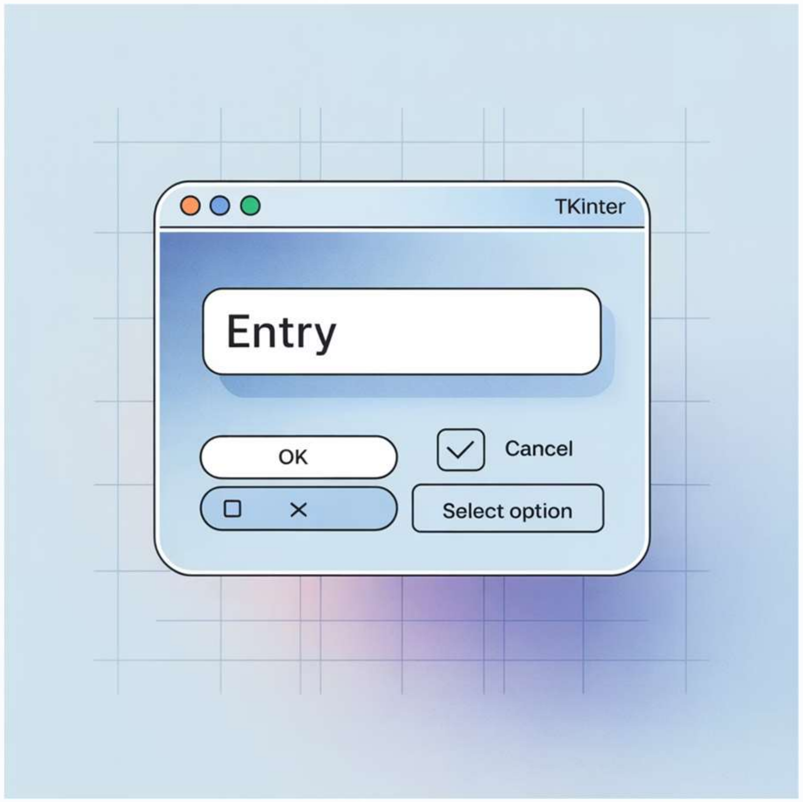
Widgets are the building blocks of any Tkinter application. Each widget is an instance of a class that represents a specific GUI element.

Common Widgets

- **Label**: Displays text or images
- **Button**: Clickable element that triggers actions
- **Entry**: Single-line text input field
- **Text**: Multi-line text input area
- **Frame**: Container to organize other widgets
- **Checkbutton:** Toggle on/off selection
- **Radiobutton**: Mutually exclusive option selection
- **Listbox**: Scrollable list of options

Each widget has its own methods and properties that control its appearance and behavior. Widgets are typically created by passing the parent container as the first argument:

```
# Create a button inside the root window
btn = tk.Button(root, text="Click Me",
bg="lightblue", fg="navy",
width=15, height=2)
btn.pack()
```



Canvas Widget

The Canvas widget provides a drawing area where you can create and manipulate graphical objects. It's one of Tkinter's most versatile widgets, allowing you to build custom visualizations and interactive graphics.

Create Basic Shapes

Draw lines, rectangles, ovals, polygons, arcs, and text on a blank surface

```
canvas = tk.Canvas(root, width=300,
height=200)
canvas.pack()
# Draw shapes
line = canvas.create_line(0, 0, 150, 150)
rect = canvas.create_rectangle(50, 25, 150,
75, fill="blue")
oval = canvas.create_oval(100, 50, 200,
100, fill="red")
```

Modify Shapes

Access and update shapes with unique IDs returned when created

```
# Change properties
canvas.itemconfig(rect, fill="green")
# Move shapes
canvas.move(oval, 10, 5) # x and y offset
# Delete shapes
canvas.delete(line)
```

Difference between Frame & Canvas

Tkinter Frame

- A container widget.
- Used to group and organize other widgets (buttons, labels, entries, etc.).
- Works with geometry managers (pack, grid, place) to arrange child widgets.
- Think of it as a section or panel in a window

Tkinter Canvas

- A drawing area.
- Used to draw graphics (shapes, lines, images, text, etc.).
- Can also hold widgets (but not its main purpose).
- Supports scrolling, moving, and animating shapes.

Frame

```
import tkinter as tk

root = tk.Tk()
root.geometry("300x200")

frame = tk.Frame(root, bg="lightblue", bd=2,
relief="sunken")
frame.pack(padx=10, pady=10, fill="both", expand=True)

label = tk.Label(frame, text="I live inside a Frame")
label.pack(pady=20)

root.mainloop()
```

Canvas

```
import tkinter as tk

root = tk.Tk()
root.geometry("300x200")

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

# Draw shapes
canvas.create_rectangle(50, 50, 150, 100, fill="lightgreen")
canvas.create_oval(180, 50, 250, 120, fill="pink")
canvas.create_text(150, 150, text="Canvas Example", font=("Arial", 12))

root.mainloop()
```

Geometry Managers

Geometry managers control how widgets are organized within a container. Tkinter offers three different layout managers, each with its own approach to widget arrangement:

1

pack()

Simplest geometry manager that arranges widgets in blocks before placing them in the parent widget.

```
button1.pack(side="top", fill="x",
padx=10, pady=5)
button2.pack(side="left")
```

2

grid()

Organizes widgets in a table-like structure defined by rows and columns.

```
label.grid(row=0, column=0)
entry.grid(row=0, column=1, padx=5,
pady=5)
button.grid(row=1, column=0,
columnspan=2)
```

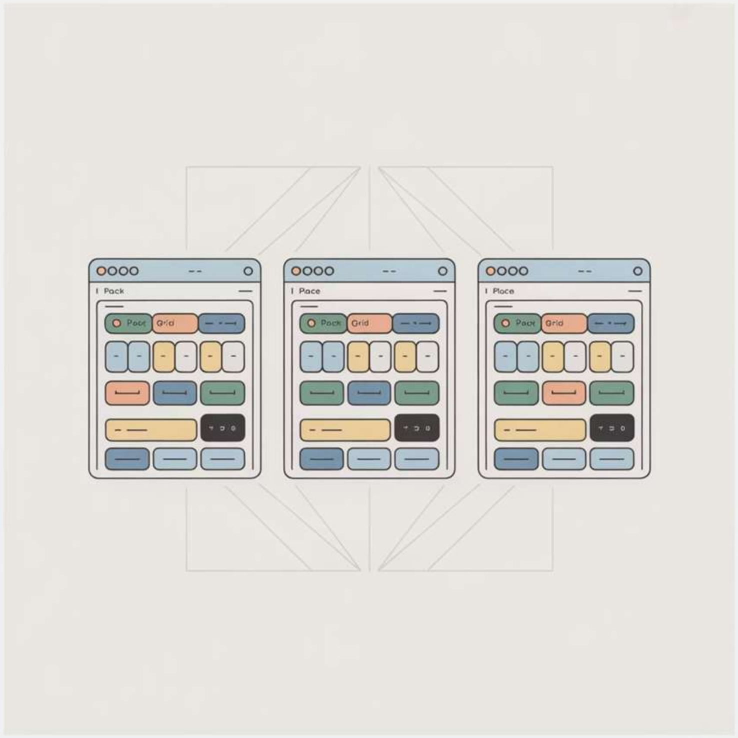
3

place()

Positions widgets using exact coordinates relative to the parent widget.

```
button.place(x=50, y=100, width=100, height=30)
```

Most Tkinter developers prefer `grid()` for complex layouts due to its flexibility and intuitive row/column structure.



⚠ **Important:** Never mix different geometry managers within the same parent container. Choose one manager for each container and stick with it.

Displaying Shapes

The Canvas widget provides a variety of methods for drawing different shapes.

Each shape-creation method returns a unique ID that you can use to modify or delete the shape later.

Common Shape Methods

- `create_line(x1, y1, x2, y2, **options)`
- `create_rectangle(x1, y1, x2, y2, **options)`
- `create_oval(x1, y1, x2, y2, **options)`
- `create_polygon(x1, y1, x2, y2, ..., **options)`
- `create_arc(x1, y1, x2, y2, **options)`
- `create_text(x, y, text="Text", **options)`

```
import tkinter as tk
root = tk.Tk()
canvas = tk.Canvas(root, width=400, height=300, bg="white")
canvas.pack(padx=10, pady=10)
# Draw a rectangle
rect = canvas.create_rectangle(
50, 50, 150, 100,
fill="#007EBD",
outline="black",
width=2
)
# Draw an oval
oval = canvas.create_oval(
200, 50, 300, 100,
fill="yellow",
outline="orange",
width=2
)
# Draw a line
line = canvas.create_line(
50, 150, 300, 200,
fill="red",
width=3,
arrow=tk.LAST
)
root.mainloop()
```

Displaying Images

Tkinter can display images through the `PhotoImage` class, which natively supports GIF and PNG formats. For other formats like JPEG, you'll need the Pillow library (PIL).

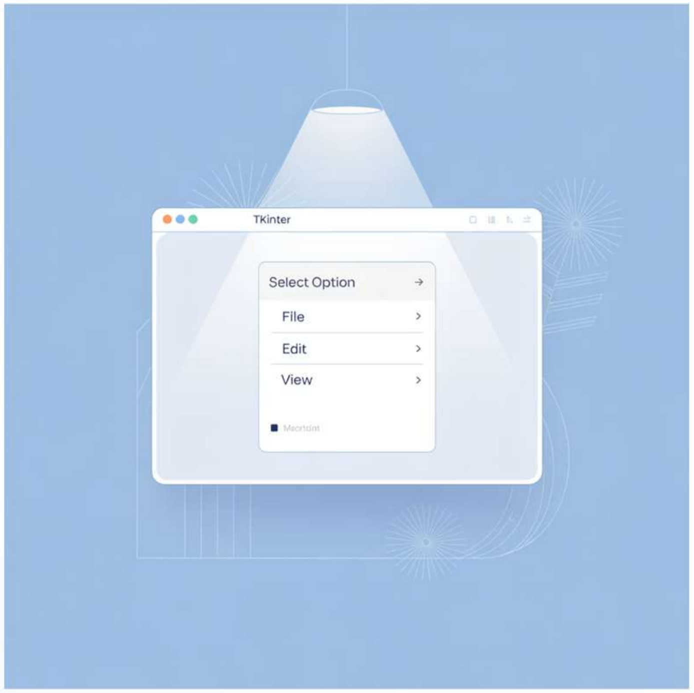
```
import tkinter as tk
from PIL import Image, ImageTk # For JPEG support

root = tk.Tk()
root.title("Image Display")
# Method 1: Native Tkinter (GIF/PNG only)
photo1 = tk.PhotoImage(file="logo.png")
label1 = tk.Label(root, image=photo1)
label1.image = photo1 # Keep a reference!
label1.pack()
# Method 2: Using Pillow (for JPEG and other formats)
pil_img = Image.open("photo.jpeg")
photo2 = ImageTk.PhotoImage(pil_img)
label2 = tk.Label(root, image=photo2)
label2.image = photo2 # Keep a reference!
label2.pack()
# Images on Canvas
canvas = tk.Canvas(root, width=400, height=300)
canvas.pack()
canvas_img = canvas.create_image(200, 150, image=photo1)
root.mainloop()
```

Menus & Popup Menus

Main Menu

Create a top-level menu bar with cascading submenus:

```
import tkinter as tk
root = tk.Tk()
root.title("Menu Example")
# Create the main menu bar
menubar = tk.Menu(root)
root.config(menu=menubar)
# Create File menu
file_menu = tk.Menu(menubar, tearoff=0)
menubar.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="New", command=lambda: print("New file"))
file_menu.add_command(label="Open", command=lambda: print("Open file"))
file_menu.add_separator()
file_menu.add_command(label="Exit", command=root.quit)
# Create Edit menu
edit_menu = tk.Menu(menubar, tearoff=0)
menubar.add_cascade(label="Edit", menu=edit_menu)
edit_menu.add_command(label="Cut", command=lambda: print("Cut"))
edit_menu.add_command(label="Copy", command=lambda: print("Copy"))
edit_menu.add_command(label="Paste", command=lambda: print("Paste"))
root.mainloop()
```



Context Menu (Right-Click)

Create a popup menu that appears on right-click:

```
def show_popup(event):
popup_menu.tk_popup(event.x_root, event.y_root)
# Create popup menu
popup_menu = tk.Menu(root, tearoff=0)
popup_menu.add_command(label="Cut", command=lambda: print("Cut"))
popup_menu.add_command(label="Copy", command=lambda: print("Copy"))
popup_menu.add_command(label="Paste", command=lambda: print("Paste"))
# Bind right-click event
root.bind("", show_popup) # Windows/Linux
# For Mac: root.bind("", show_popup)
```

Menus enhance usability by providing a familiar interface for users to access application functionality.

Mouse & Key Events

Tkinter's event binding system allows you to capture and respond to various user interactions, creating dynamic and responsive applications.

Mouse Events

- `<Button-1>` - Left mouse button
- `<Button-2>` - Middle mouse button
- `<Button-3>` - Right mouse button
- `<ButtonRelease-1>` - Left button release
- `<Double-Button-1>` - Double-click
- `<B1-Motion>` - Mouse drag with left button
- `<Enter>` - Mouse enters widget
- `<Leave>` - Mouse leaves widget

```
def on_click(event):
print(f"Clicked at: {event.x}, {event.y}")
canvas.bind("", on_click)
```

Keyboard Events

- `<KeyPress>` or `<Key>` - Any key
- `<KeyPress-a>` - Specific key (letter 'a')
- `<KeyRelease>` - Key release
- `<Return>`, `<Escape>` - Special keys
- `<Shift-Up>` - Modifier combinations

```
def key_pressed(event):
print(f"Key: {event.char}, Keycode: {event.keycode}")
print(f"State: {event.state}") # Modifier keys
root.bind("", key_pressed)
```

The `event` object contains information about the event, including coordinates for mouse events and character/keycode information for keyboard events.

Scrollbars

Scrollbars allow users to navigate through content that exceeds the visible area of a widget. They're commonly used with Text, Canvas, and Listbox widgets.

Creating Scrollable Widgets

To add scrolling capability:

1. Create both the widget and its scrollbar
2. Connect them using the `yscrollcommand` and `command` parameters
3. Arrange them side by side using a geometry manager

For bidirectional scrolling (both horizontal and vertical), you'll need to create and configure two scrollbars.

```
from tkinter import *
root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack(side=RIGHT, fill=Y)
mylist = Listbox(root, yscrollcommand=scrollbar.set)

for line in range(100):
   mylist.insert(END, 'This is line number' + str(line))
mylist.pack(side=LEFT, fill=BOTH)
scrollbar.config(command=mylist.yview)
mainloop()
```

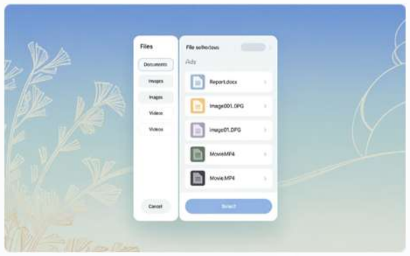
Standard Dialog Boxes

Tkinter provides several built-in dialog boxes that follow the operating system's native look and feel, saving you from having to create common dialogs from scratch.

Message Dialogs

Display information, warnings, and errors to users:

```
from tkinter import messagebox# Informationmessagebox.showinfo("Success", "File saved successfully!")# Warningmessagebox.showwarning("Warning", "Low disk space")# Errormessagebox.showerror("Error", "Could not connect to server")# Yes/No questionif messagebox.askyesno("Confirm", "Are you sure?"):    print("User selected Yes")
```



File Dialogs

Allow users to select files or directories:

```
from tkinter import filedialog# Open filefile_path = filedialog.askopenfilename(    title="Select a file",    filetypes=(("Text files", "*.txt"), ("All files", "*.*")))# Save filesave_path = filedialog.asksaveasfilename(    defaultextension=".txt")# Select directorydir_path = filedialog.askdirectory()
```

Color Chooser

Let users select colors for your application:

```
from tkinter import colorchoosercolor_rgb, color_hex = colorchooser.askcolor(    title="Select color",    initialcolor="#007EBD")if color_hex:  # None if user cancels    label.config(bg=color_hex)    print(f"RGB: {color_rgb}, Hex: {color_hex}")
```

These dialog boxes help create a more polished, professional application with minimal code. They automatically adapt to the user's operating system, providing a native look and feel.

Radio Button, Checkbox, List Box

Tkinter provides several built-in widgets like radio button, check box, list box

Radio Button

It allows user to select one option from a set of choices. They are grouped by sharing the same variable.

```
from tkinter import *
root = Tk()
v = IntVar()
Radiobutton(root, text='GfG', variable=v,
value=1).pack(anchor=W)
Radiobutton(root, text='MIT', variable=v,
value=2).pack(anchor=W)
mainloop()
```

Check Box

A checkbox can be toggled on or off.
It can be linked to a variable to store its state.

```
from tkinter import *
master = Tk()
var1 = IntVar()
Checkbutton(master, text='male',
variable=var1).grid(row=0, sticky=W)
var2 = IntVar()
Checkbutton(master, text='female',
variable=var2).grid(row=1, sticky=W)
mainloop()
```

List Box

It displays a list of items from which a user can select one or more.

```
from tkinter import *
top = Tk()
Lb = Listbox(top)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
top.mainloop()
```

Combobox

The Combobox widget from tkinter.ttk is created using Combobox class. Its options are set via values parameter, and a default value is assigned using set() method. An event handler (like on_select) can be bound using bind() to update other widgets based on selected item.

```
import tkinter as tk
from tkinter import ttk

def select(event):
    selected_item = combo_box.get()
    label.config(text="Selected Item: " + selected_item)

root = tk.Tk()
root.title("Combobox Example")

# Create a label
label = tk.Label(root, text="Selected Item: ")
label.pack(pady=10)

# Create a Combobox widget
combo_box = ttk.Combobox(root, values=["Option 1", "Option 2", "Option 3"], state='readonly'))
combo_box.pack(pady=5)

# Set default value
combo_box.set("Option 1")

# Bind event to selection
combo_box.bind("<<ComboboxSelected>>", select)
root.mainloop()
```

Progressbar

Progressbar indicates the progress of a long-running task. When the button is clicked, the progressbar fills up to 100% over a short period, simulating a task that takes time to complete.

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

def start_progress():
    progress.start()

    # Simulate a task that takes time to complete
    for i in range(101):
      # Simulate some work
        time.sleep(0.05)
        progress['value'] = i
        # Update the GUI
        root.update_idletasks()
    progress.stop()

root = tk.Tk()
root.title("Progressbar Example")

# Create a progressbar widget
progress = ttk.Progressbar(root, orient="horizontal", length=300, mode="determinate")
progress.pack(pady=20)

# Button to start progress
start_button = tk.Button(root, text="Start Progress", command=start_progress)
start_button.pack(pady=10)
root.mainloop()
```

Animations

Creating animations in Tkinter involves updating the position or appearance of Canvas items at regular intervals. This is achieved using the `after()` method to schedule repeated function calls.

```
import tkinter as tk
root = tk.Tk()
canvas = tk.Canvas(root, width=400, height=300, bg="black")
canvas.pack()
# Create a ball
ball = canvas.create_oval(10, 10, 30, 30, fill="#007EBD")
x_speed, y_speed = 4, 3  # Movement speed in pixels
def animate():
    # Get current ball position
    pos = canvas.coords(ball)
    print(pos)
    print()
    # Check for collisions with walls
    if pos[0] <= 0 or pos[2] >= 400:
        global x_speed
        x_speed = -x_speed
    if pos[1] <= 0 or pos[3] >= 300:
        global y_speed
        y_speed = -y_speed

    # Move the ball
    canvas.move(ball, x_speed, y_speed)

    # Schedule the next animation frame
    root.after(30, animate)  # 30ms = ~33 FPS

# Start animation
animate()
root.mainloop()
```

For complex animations or games, you might want to consider dedicated game libraries like Pygame. However, Tkinter is suitable for simple animations and educational game projects.