Container Widgets, Geometry Managers, and Canvas

01219116 Computer Programming II

Chaiporn Jaikaeo

Department of Computer Engineering
Kasetsart University

Materials adapted from TkDocs Tutorial (https://tkdocs.com/tutorial/index.html) Revised 2021-02-08

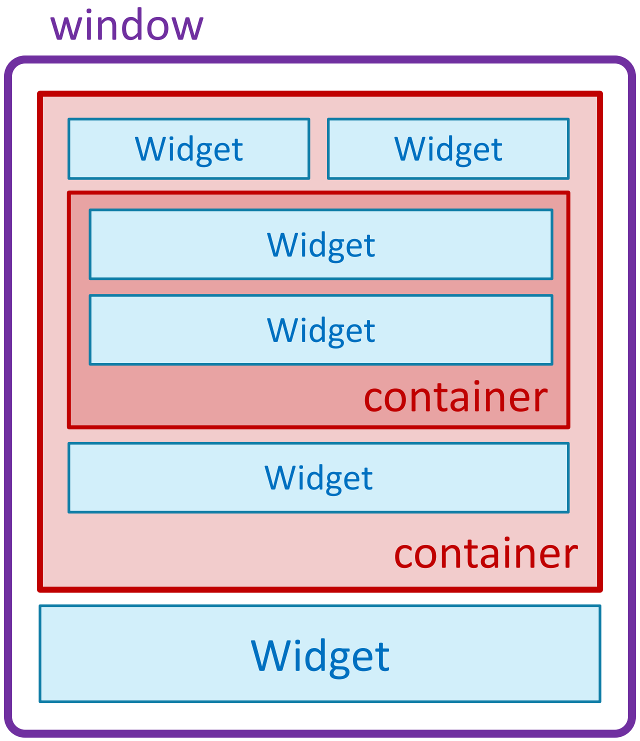
Outline

- Container widgets
- Geometry managers
- Drawing canvas
- Integration with turtle graphics and Matplotlib

Container Widgets

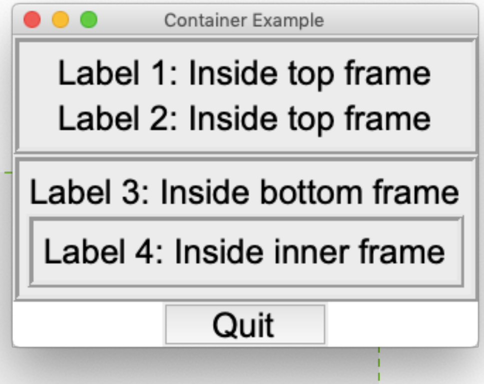
Container Widgets

- **Container widgets** can contain other widgets
  - E.g., TopLevel, Frame, LabelFrame, PanedWindow, Notebook
- A container can contain other containers, each of which can contain other containers, and so on
- A window, e.g., the root window, is considered a top-level container



Example: Nested Widgets

- This example demonstrates use of Frame widgets to contain other widgets



Label 1: Inside top frame
Label 2: Inside top frame

Label 3: Inside bottom frame

Label 4: Inside inner frame

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.title("Container Example")
style = ttk.Style()
style.configure(".", font=("Arial",24))
top = ttk.Frame(name="top", borderwidth=5, padding=5, relief="ridge")
bottom = ttk.Frame(name="bottom", borderwidth=5, padding=5, relief="ridge")
inner = ttk.Frame(bottom, name="inner", borderwidth=5, padding=5, relief="ridge")
ttk.Label(top, name="label1", text="Label 1: Inside top frame").pack()
ttk.Label(top, name="label2", text="Label 2: Inside top frame").pack()
ttk.Label(bottom, name="label3", text="Label 3: Inside bottom frame").pack()
ttk.Label(inner, name="label4", text="Label 4: Inside inner frame").pack()
btn_quit = ttk.Button(name="quit", text="Quit", command=root.destroy)
top.pack(fill=tk.BOTH)
bottom.pack(fill=tk.BOTH)
inner.pack(fill=tk.BOTH)
btn_quit.pack()

root.mainloop()
```

A parent container can be specified as the first argument during widget creation

Inspecting Widget Hierarchy

- Use the following recursive function to dump the widget hierarchy, starting at the root window

```
def dump_widget(widget, indent=0):
    print((" "*indent) + str(widget))
    for w in widget.winfo_children():
        dump_widget(w, indent+2)

dump_widget(root)
```

- The result looks like

```
.
  .top
    .top.label1
    .top.label2
  .bottom
    .bottom.inner
      .bottom.inner.label4
    .bottom.label3
  .quit
```

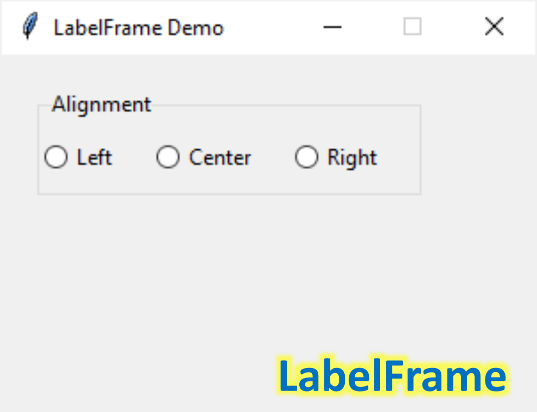
Modifying the Application Boilerplate

- Previously, our application class inherits directly from the **Tk** class
- Let's modify the code so that it inherits from Frame instead
  - It will be easily integrated into another "umbrella" application

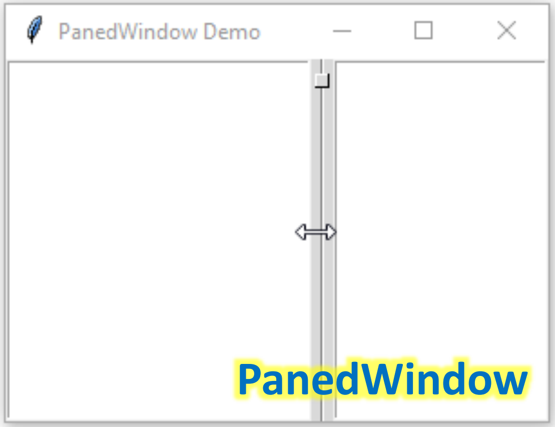
```
import tkinter as tk

class App(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("My App")
        :

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

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        :

if __name__ == "__main__":
    root = tk.Tk()
    root.title("My App")
    app = App(root)
    root.mainloop()
```

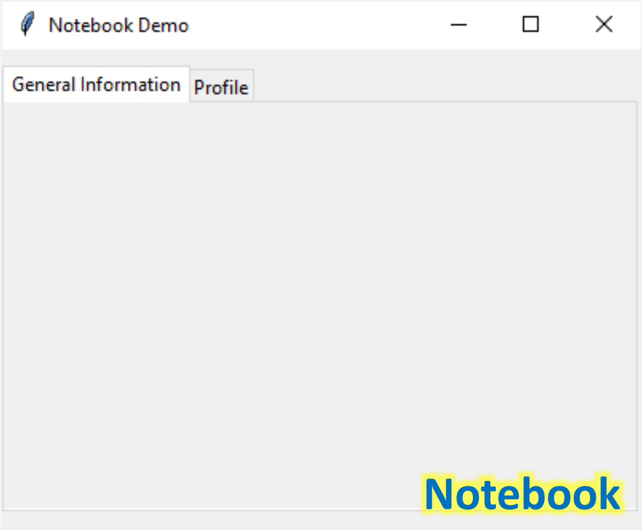
Some Common Containers



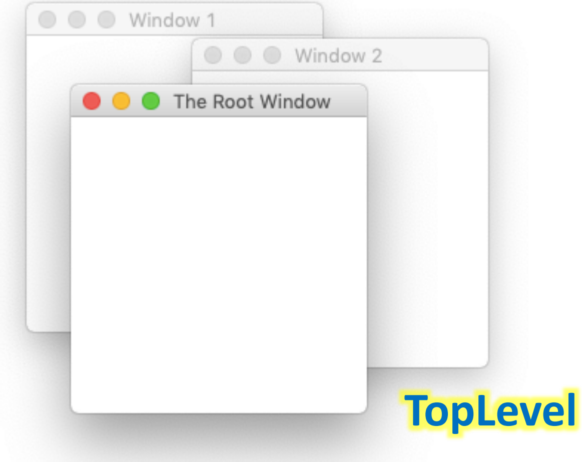
LabelFrame



PanedWindow



Notebook



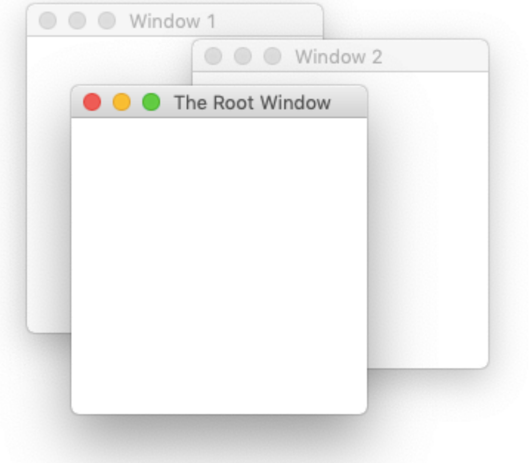
TopLevel

https://www.pythontutorial.net/tkinter/

TopLevel Widget

- TopLevel widget provides its own window container
- Unlike other widgets, TopLevel window appears right away without a geometry manager

```
import tkinter as tk

root = tk.Tk()
root.title("The Root Window")
win1 = tk.Toplevel()
win1.title("Window 1")
win2 = tk.Toplevel()
win2.title("Window 2")

root.mainloop()
```



- Use the **destroy()** method to destroy the window, as well as its contained widgets

Geometry Managers

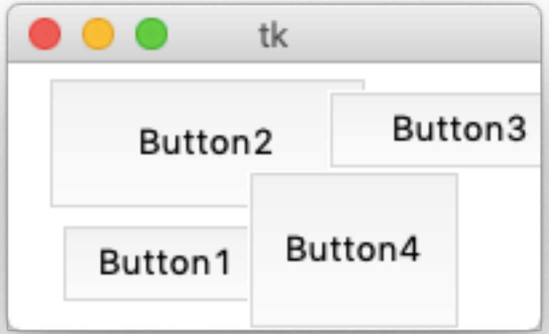
Geometry Managers

- A geometry manager is responsible for placing widgets in a container
- So far, only the **pack** geometry manager has been used
  - Its behavior is often hard to understand
  - Order of packed widgets determines layout, modifying existing layouts can be difficult
  - Aligning widgets in different parts of the user interface is also much trickier

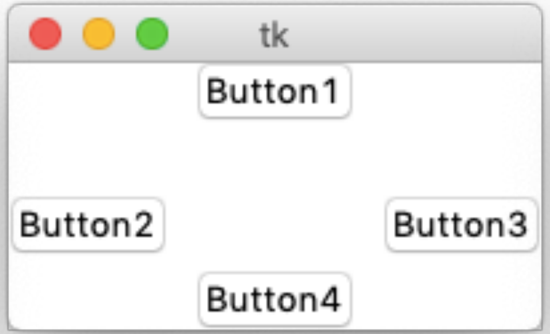
Tk's Geometry Managers

- The **place** manager
  - Provides absolute positioning and sizing
  - Very hard to create multi-platform applications
- The **pack** manager
  - Organizes widgets in horizontal and vertical boxes
  - Controls the layout with fill, expand, and side options
- The **grid** manager
  - Places widgets in a two-dimensional grid
  - Similar to HTML tables

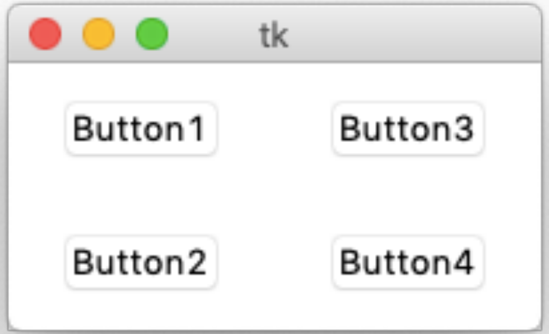
Geometry Managers: Examples



"place" manager



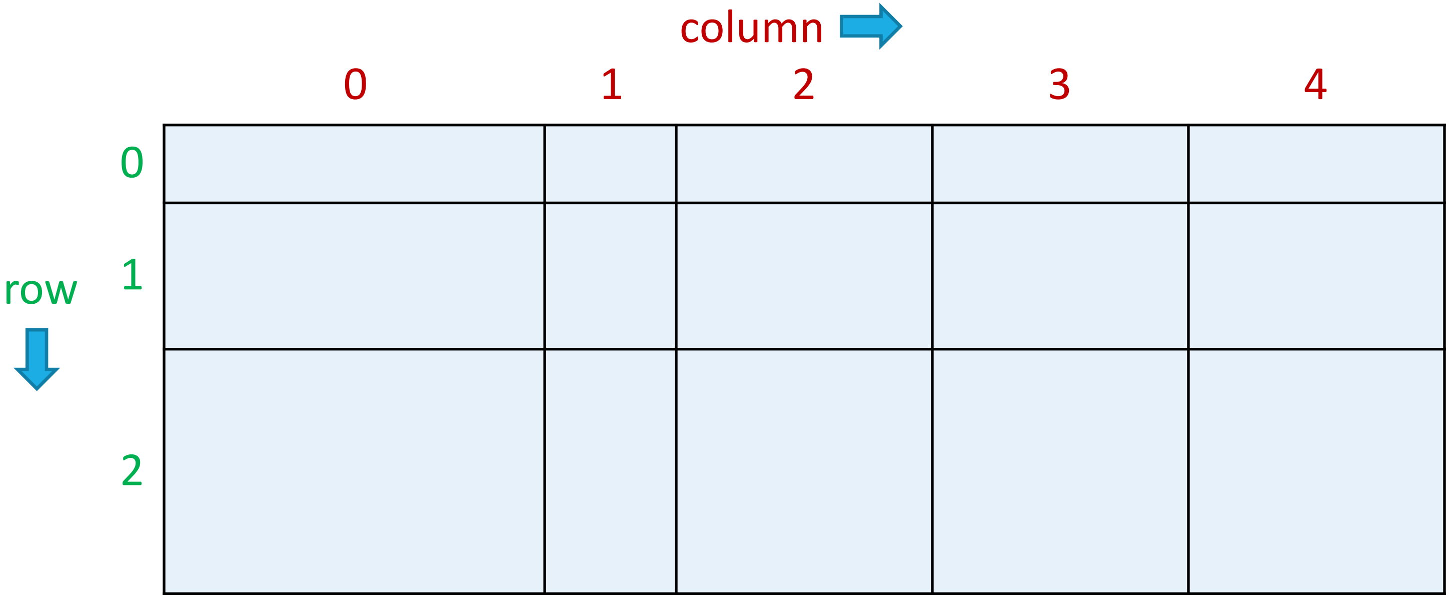
"pack" manager



"grid" manager

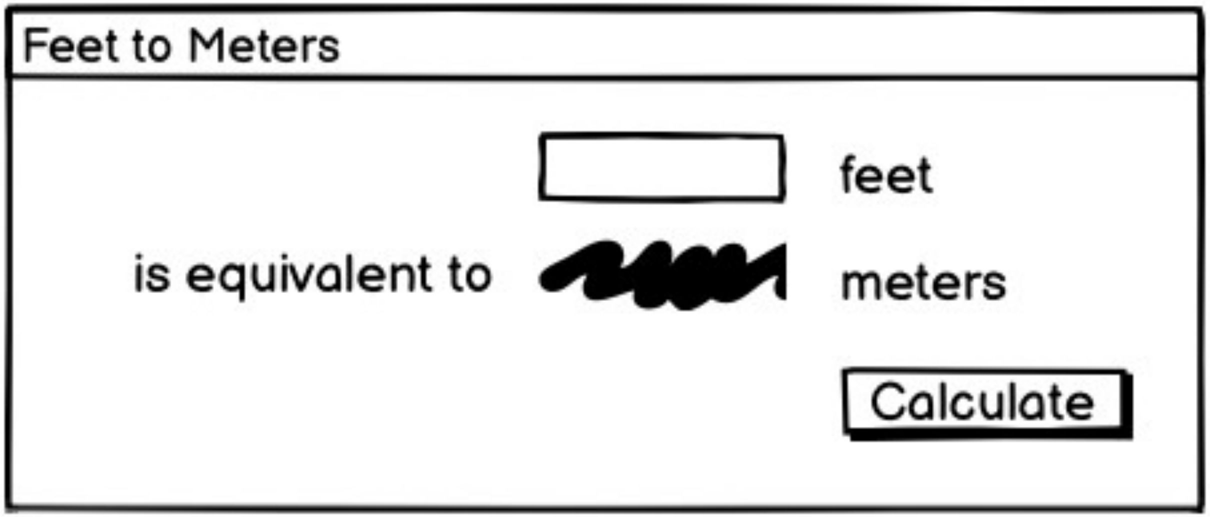
Grid Geometry Manager

- Best choice for general use
- Easy to arrange multiple widgets with predictable results
- Each widget is assigned a column number and a row number



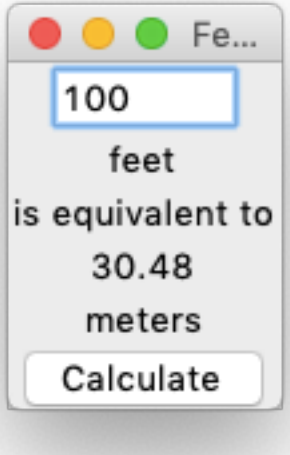
Example: Feet-to-Meters Converter

- Let's build a simple unit conversion application



Feet-to-Meters with pack Manager

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

class App(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        self.mainframe = ttk.Frame(root)
        self.mainframe.pack()

        self.feet = tk.StringVar()
        self.meters = tk.StringVar()
        self.feet_entry = ttk.Entry(self.mainframe, width=7, textvariable=self.feet)
        self.feet_entry.pack()
        ttk.Label(self.mainframe, text="feet").pack()
        ttk.Label(self.mainframe, text="is equivalent to").pack()
        ttk.Label(self.mainframe, textvariable=self.meters).pack()
        ttk.Label(self.mainframe, text="meters").pack()
        ttk.Button(self.mainframe, text="Calculate", command=self.calculate).pack()

        self.feet_entry.focus()
        self.feet_entry.bind("<Return>", self.calculate)

    def calculate(self, *args):
        try:
            value = float(self.feet.get())
            self.meters.set(int(0.3048 * value * 10000.0 + 0.5)/10000.0)
        except ValueError:
            pass

if __name__ == "__main__":
    root = tk.Tk()
    root.title("Feet to Meters")
    app = App(root)
    root.mainloop()
```



100

feet

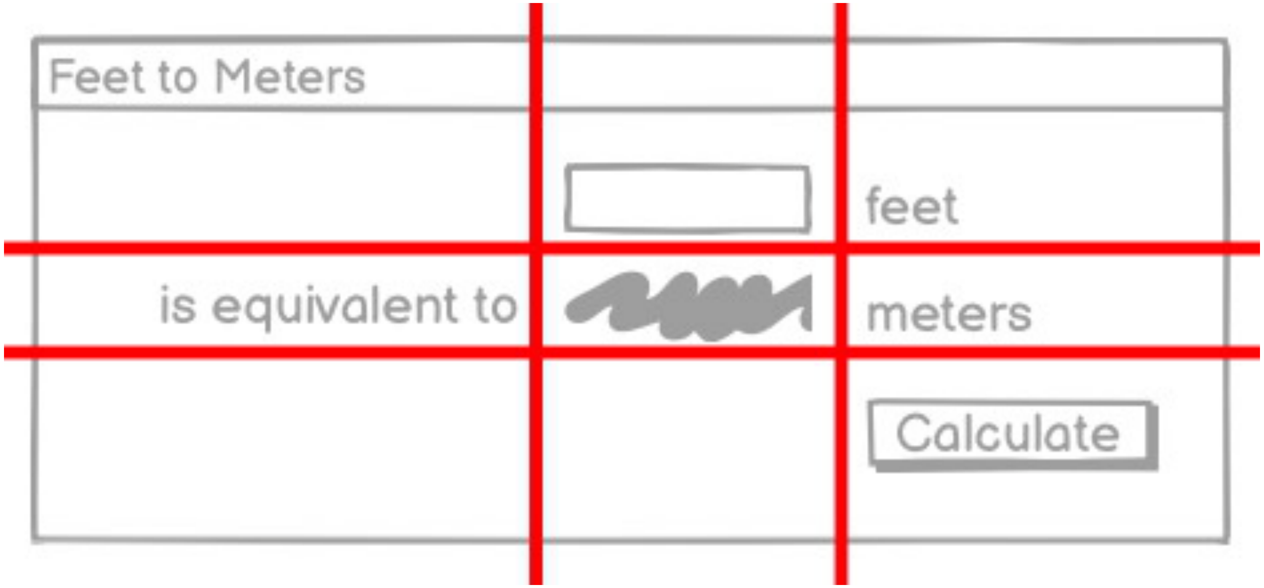
is equivalent to

30.48

meters

Feet-to-Meters: Grid Layout

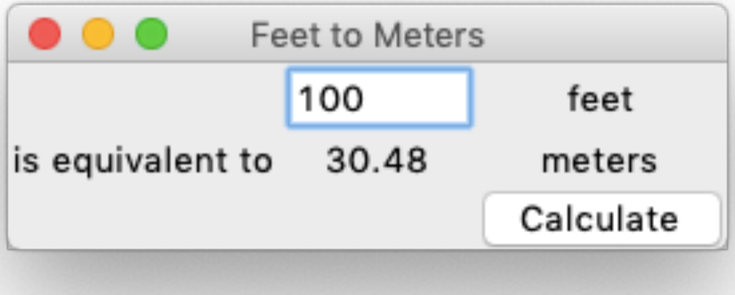
- Widgets can be arranged into a 3x3 grid



Feet-to-Meters: Switch to grid

- Replace all calls to **pack()** with **grid()**

```
class App(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        self.mainframe = ttk.Frame(root)
        self.mainframe.grid(column=0, row=0)

        self.feet = tk.StringVar()
        self.meters = tk.StringVar()
        self.feet_entry = ttk.Entry(self.mainframe, width=7, textvariable=self.feet)
        self.feet_entry.grid(column=1, row=0)
        ttk.Label(self.mainframe, text="feet").grid(column=2, row=0)
        ttk.Label(self.mainframe, text="is equivalent to").grid(column=0, row=1)
        ttk.Label(self.mainframe, textvariable=self.meters).grid(column=1, row=1)
        ttk.Label(self.mainframe, text="meters").grid(column=2, row=1)
        ttk.Button(self.mainframe, text="Calculate", command=self.calculate).grid(column=2, row=2)
```



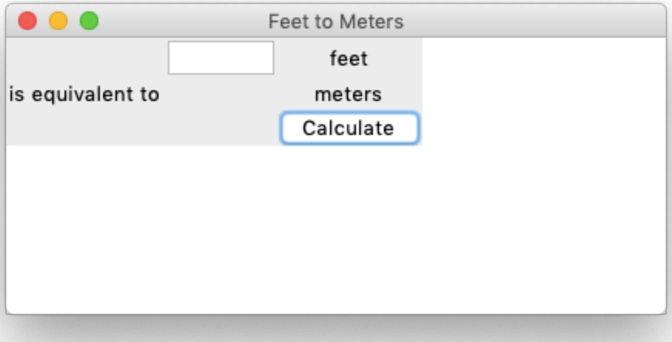
Feet to Meters

100 feet

is equivalent to 30.48 meters

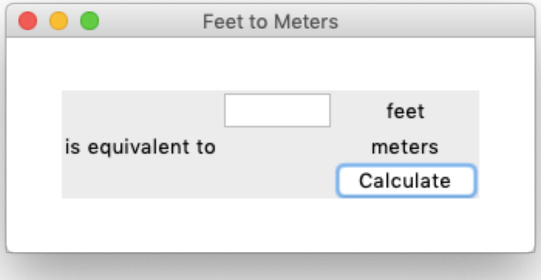
Resizing Rows and Columns

- The current Feet-to-Meters app does not get resized properly



- Adjust the amount of space added to each row or column relative to other rows and columns by calling the parent's **`rowconfigure()`** and **`columnconfigure()`** methods

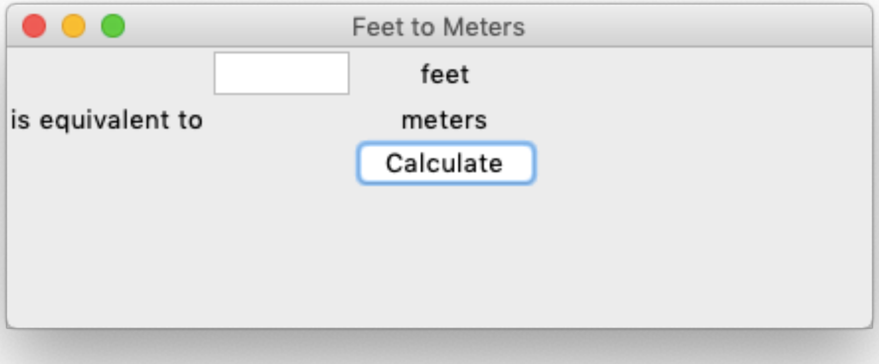
```
parent.rowconfigure(0, weight=1)
parent.columnconfigure(0, weight=1)
```



The **sticky** Option

- The Frame widget still does not get resized properly because its edges do not stick to the container's edges
- Use the **sticky** option with compass letters (N,E,W,S) to make them sticky at the specified directions

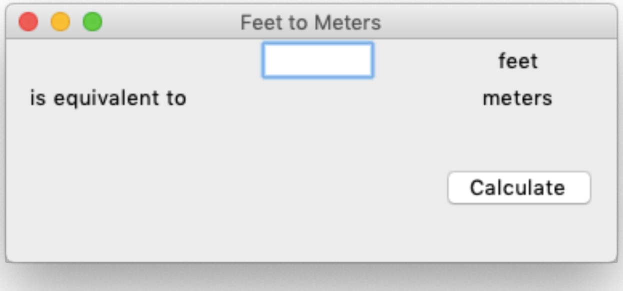
```
self.mainframe.grid(column=0, row=0, sticky="NEWS")
```



Resizing Inner Grid

- The inner 3x3 grid also needs to be configured to get resized properly
  - All columns should expand along with the window's width, with a greater weight for the middle column
  - Only the last row should expand with the window's height

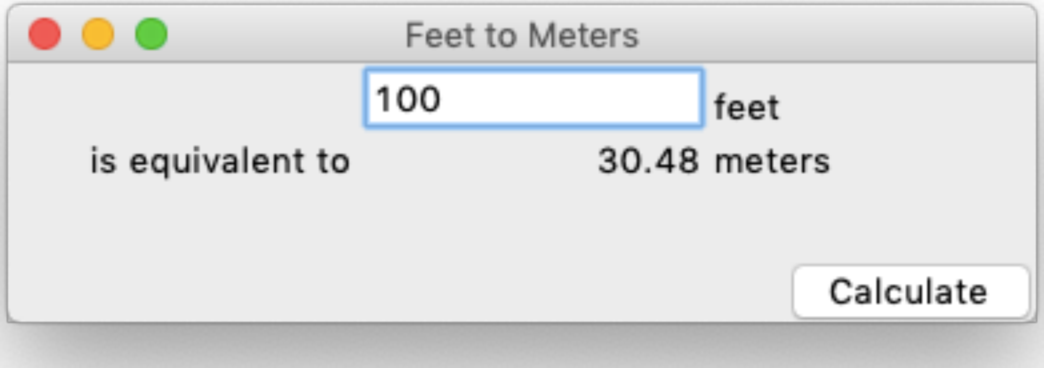
```
self.mainframe.rowconfigure(2, weight=1)
self.mainframe.columnconfigure(0, weight=1)
self.mainframe.columnconfigure(1, weight=2)
self.mainframe.columnconfigure(2, weight=1)
```



Making Inner Widgets Sticky

- Inner widgets should be made sticky to certain edges

```
self.feet_entry.grid(column=1, row=0, sticky="WE")
ttk.Label(self.mainframe, text="feet").grid(column=2, row=0, sticky="SW")
ttk.Label(self.mainframe, text="is equivalent to").grid(column=0, row=1, sticky="NE")
ttk.Label(self.mainframe, textvariable=self.meters).grid(column=1, row=1, sticky="NE")
ttk.Label(self.mainframe, text="meters").grid(column=2, row=1, sticky="NW")
ttk.Button(self.mainframe, text="Calculate", command=self.calculate).grid(column=2, row=2, sticky="SE")
```



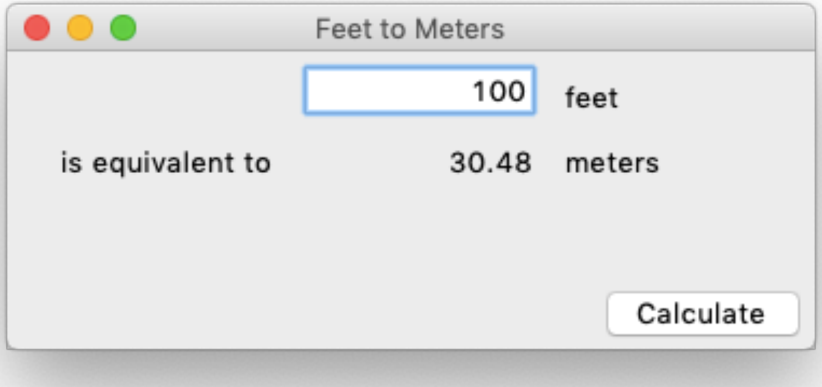
Finishing Touches

- A few more things can be polished
  - The number entry should be right-aligned

```
self.feet_entry = ttk.Entry(self.mainframe, width=7,
                            textvariable=self.feet, justify="right")
```

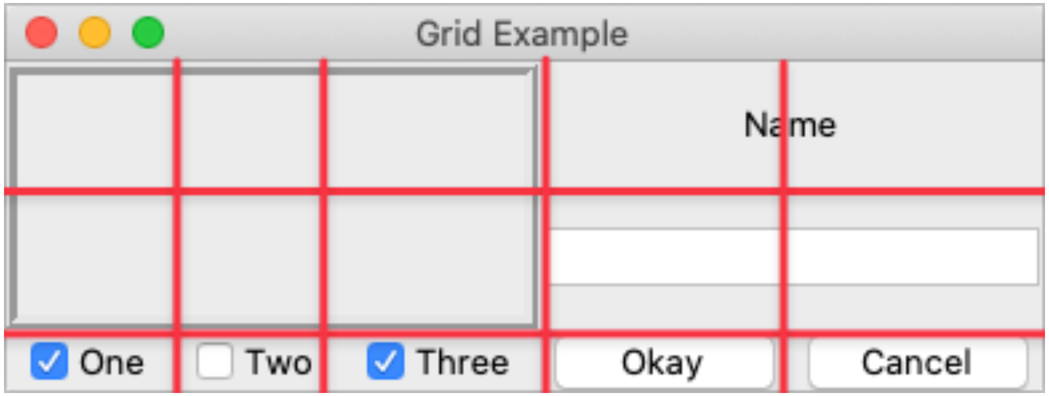
  - Paddings around all the inner widgets

```
for child in self.mainframe.winfo_children():
    child.grid_configure(padx=5, pady=5)
```



Spanning Multiple Cells

- Use **columnspan** and **rowspan** options for widgets that take up more than single cells

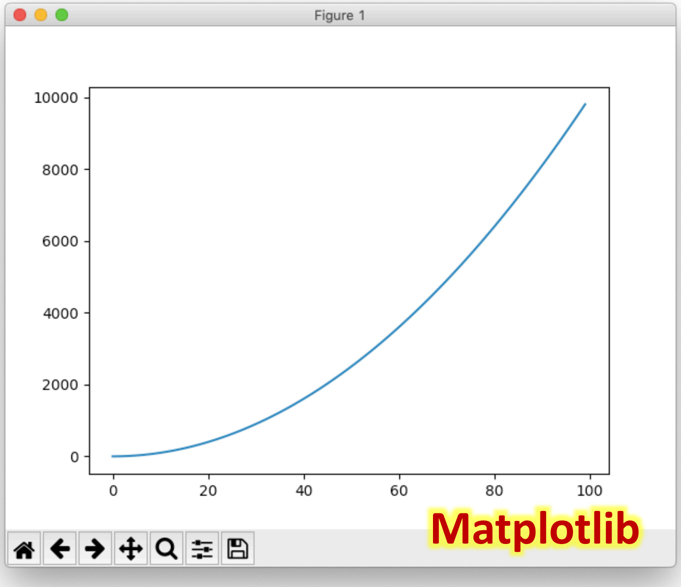


☑ One □ Two ☑ Three

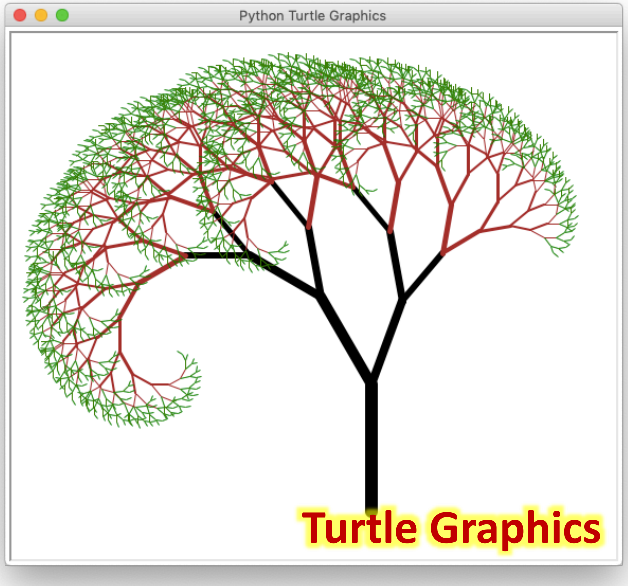
Drawing Canvas

Canvas Widget

- A Canvas widget manages a 2D collection of graphical objects
  - E.g., lines, circles, text, images
- Many libraries employ canvas widgets to create drawings



Matplotlib

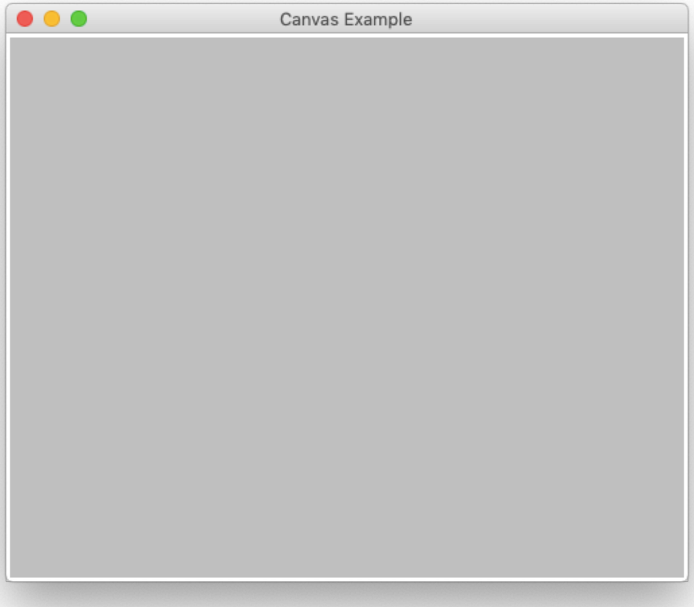


Turtle Graphics

Creating a Canvas

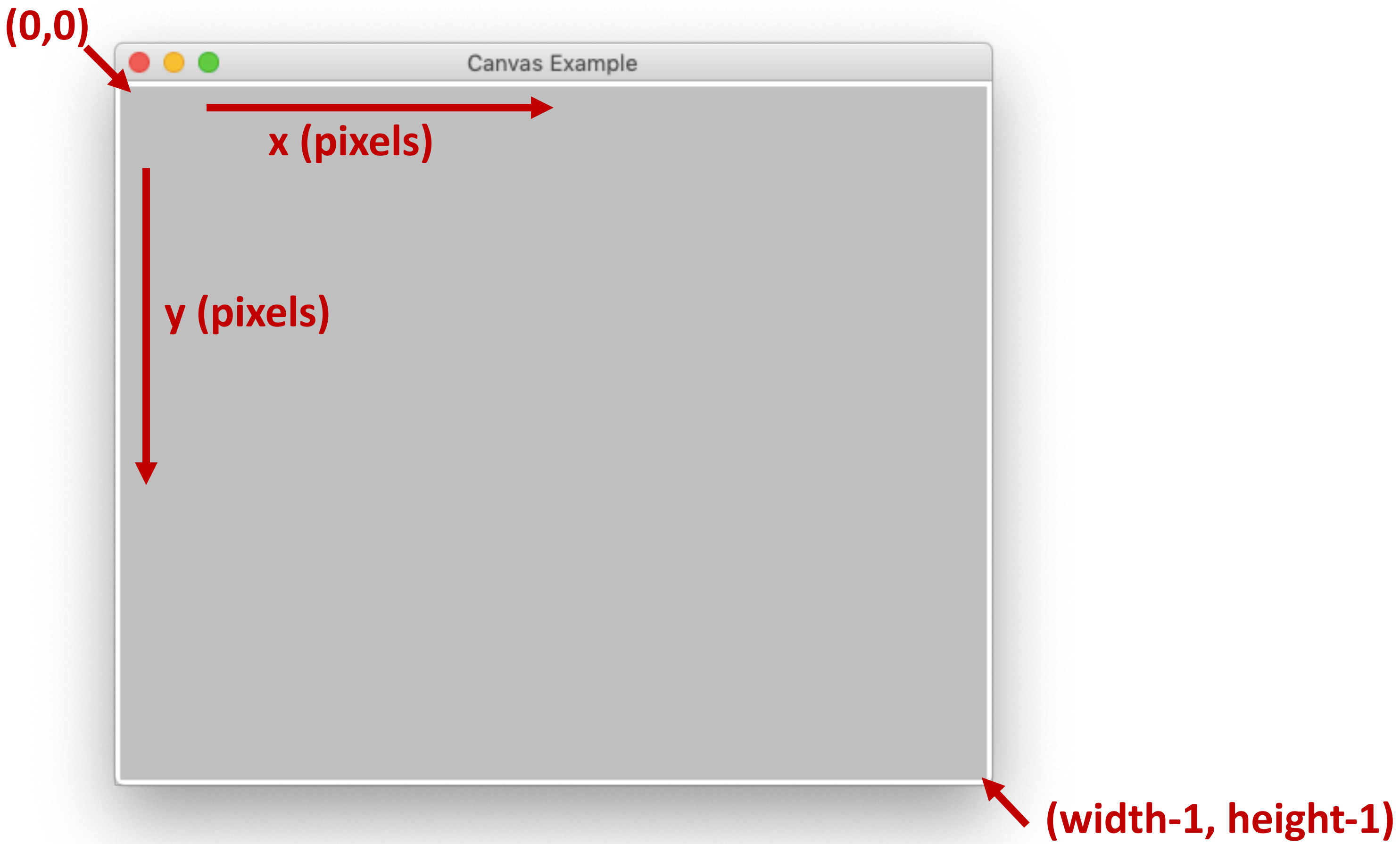
- Use the **Canvas** class to create a canvas widget (only available in the classic tk module, not ttk)
- E.g.,

```
import tkinter as tk

root = tk.Tk()
root.title("Canvas Example")
frame = tk.Frame()
frame.pack()
canvas = tk.Canvas(frame,
                   width=500, height=400,
                   background='gray75')
canvas.pack()

root.mainloop()
```



Canvas Coordinate System

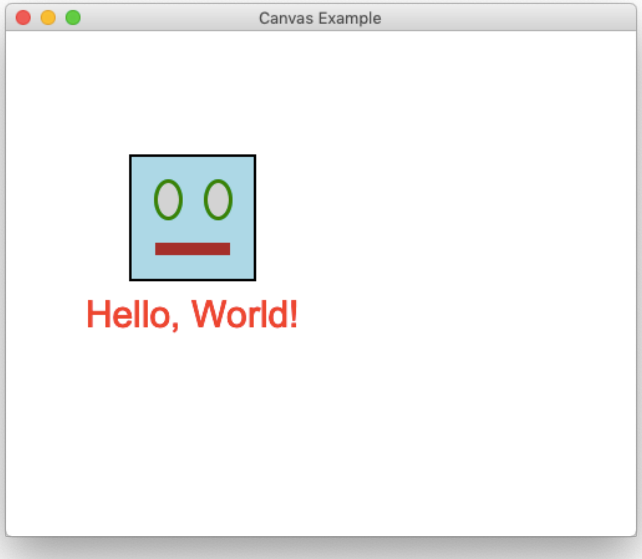
- The top-left corner, known as the ***origin***, has coordinates (0,0)



Canvas Items

- Use various create_* methods to create graphical items

```
canvas.create_rectangle(100, 100, 200, 200,
    width=2, fill="lightblue")

canvas.create_oval(120, 120, 140, 150,
    width=3, fill="lightgray", outline="green")

canvas.create_oval(160, 120, 180, 150,
    width=3, fill="lightgray", outline="green")

canvas.create_line(120, 175, 180, 175,
    width=10, fill="brown")

canvas.create_text(150, 210, text='Hello, World!',
    anchor='n', fill='red', font=("Arial",30))
```



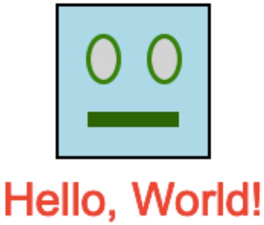
Modifying Items

- Each call to **create_xxx()** returns an ID

```
mouth = canvas.create_line(120, 175, 180, 175, width=10, fill="brown")
```

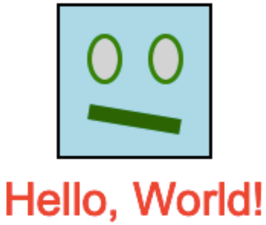
- Use canvas's **itemconfigure()** method to reconfigure an item

```
canvas.itemconfigure(mouth, fill="darkgreen")
```



- Use canvas's **coords()** method to reconfigure an item

```
canvas.coords(mouth, 120, 170, 180, 180)
```

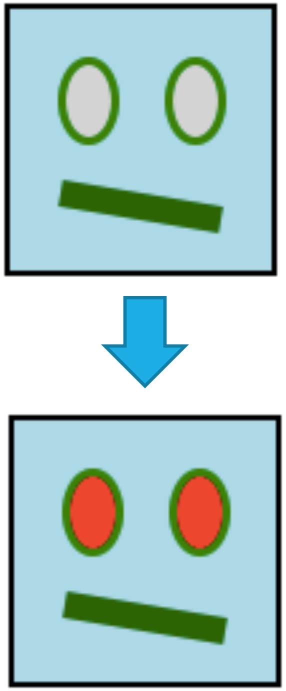


Tags

- A tag is used to address a group of canvas items
  - More convenient to reconfigure a group of items all at once

```
canvas.create_oval(120, 120, 140, 150,
    width=3, fill="lightgray", outline="green",
    tags=['eyes'])
canvas.create_oval(160, 120, 180, 150,
    width=3, fill="lightgray", outline="green",
    tags=['eyes'])
```

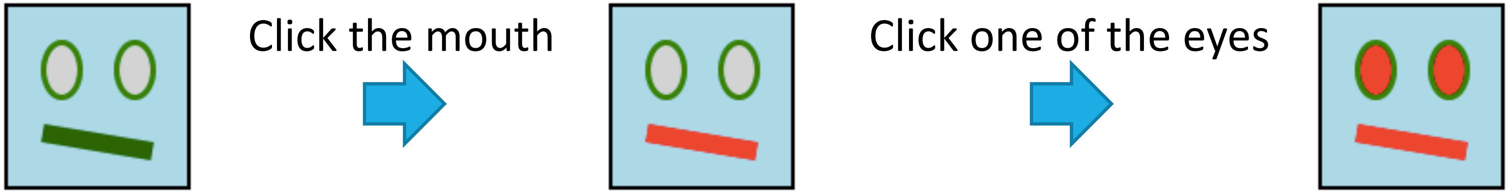
```
canvas.itemconfigure('eyes', fill="red")
```



Event Bindings

- Individual canvas items can also trigger events
- Use the canvas's **tag_bind()** method to assign an event handler to an item or a group of items
    - The first argument can be an individual item's ID or a tag

```
canvas.tag_bind(mouth, "<Button-1>",
        lambda e: canvas.itemconfigure(mouth, fill="red"))

canvas.tag_bind('eyes', "<Button-1>",
        lambda e: canvas.itemconfigure('eyes', fill="red"))
```



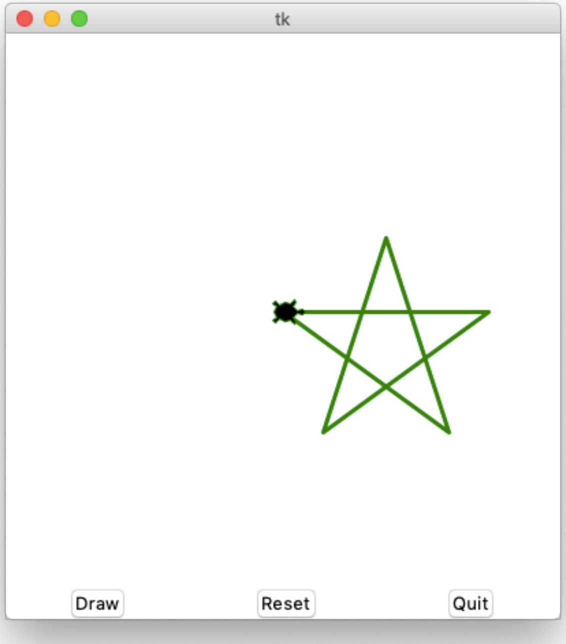
Click the mouth

Click one of the eyes

Integration with Turtle Graphics

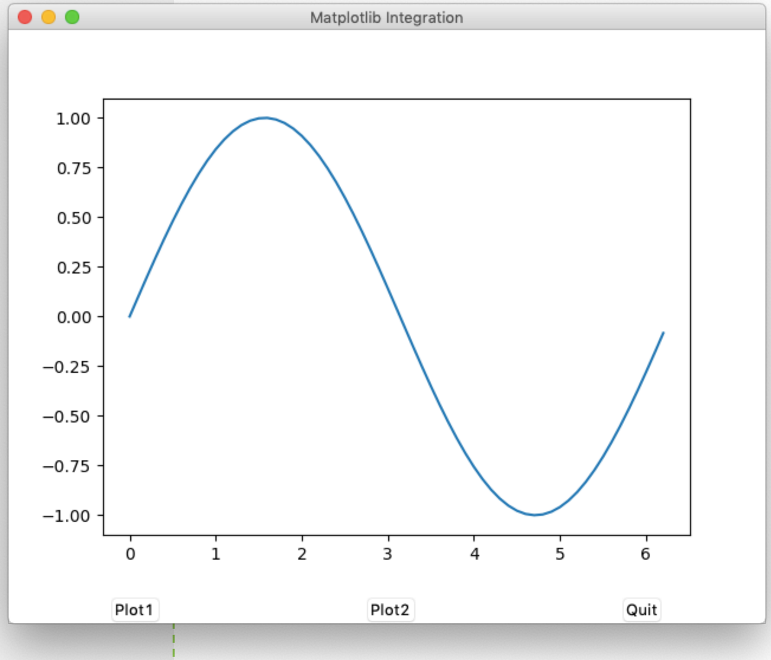
- The Python Turtle Graphics library can use an existing canvas to display its turtle(s) created with **`RawTurtle`** class
- Turtle instantiation will overwrite anything on the canvas and change its coordinate system so that (0,0) is located at the center

```
import turtle
import tkinter as tk

def draw():
    for i in range(5):
        turtle.forward(150)
        turtle.right(144)

root = tk.Tk()
canvas = tk.Canvas(width=400, height=400, bg="black")
canvas.grid(column=0,row=0,columnspan=3)
turtle = turtle.RawTurtle(canvas)
turtle.pencolor("green")
turtle.width(3)
turtle.shape("turtle")
tk.Button(text="Draw", command=draw).grid(column=0,row=1)
tk.Button(text="Reset", command=turtle.reset).grid(column=1,row=1)
tk.Button(text="Quit", command=root.destroy).grid(column=2,row=1)

root.mainloop()
```



Integration with Matplotlib

```
import tkinter as tk
import matplotlib
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
import numpy as np

def plot1(canvas, axes):
    axes.clear()
    x = np.arange(100)
    axes.plot(x,x**2)
    canvas.draw()

def plot2(canvas, axes):
    axes.clear()
    x = np.arange(0,2*np.pi,0.1)
    axes.plot(x,np.sin(x))
    canvas.draw()

root = tk.Tk()
root.title("Matplotlib Integration")

# create Matplotlib figure and plotting axes
fig = Figure()
ax = fig.add_subplot()

# create a canvas to host the figure and place it into the root window
canvas = FigureCanvasTkAgg(fig, master=root)
canvas.get_tk_widget().grid(column=0, row=0, columnspan=3)

# create a few action buttons
tk.Button(text="Plot1", command=lambda: plot1(canvas,ax)).grid(column=0, row=1)
tk.Button(text="Plot2", command=lambda: plot2(canvas,ax)).grid(column=1, row=1)
tk.Button(text="Quit", command=root.destroy).grid(column=2, row=1)

root.mainloop()
```



Conclusion

- Complex applications require widgets to be combined into groups and nicely placed on screen
  - Containers are special widgets that group widgets together
  - Geometry managers are responsible for placing widgets over their container. Available geometry managers are **pack**, **place**, and **grid**
- Canvas widgets allow creation of 2D graphics
- Several Python libraries, such as Matplotlib and Turtle graphics, utilize Tkinter's canvas, which can be integrated into our GUI applications