CIS 41B
Advanced Python Programming

GUI

De Anza College
Instructor: Clare Nguyen

GUI and Tk

- GUI is short for **g**raphical **u**ser **i**nterface. This interface provides a way for the user to interact with an electronic device through graphical elements such as windows, scroll bars, pull down menus, buttons, etc.
- GUI is how most modern applications interact with the user. It is considered to be more user friendly than the classic text based interface, where the user enters one text string command at a time. (This is known as CLI or command line interface.)
- When creating a GUI for an application, we can use one of many object oriented libraries with a variety of built-in classes so that we don't have to "reinvent the wheel" from scratch.
- A popular and classic library for GUI is the Tk library, written in C:
  - It provides many GUI parent classes, where each class is a graphical element such as a window class, button class, menu class, etc.
  - The Tk library can be used with different programming languages such as C++, Perl, Ruby, Python, etc.
  - When the Tk objects run, they interact with the host system's windowing infrastructure. This means our application's GUI will look like a Windows application when it runs on Windows, and it will look like a Mac application when it runs on the Mac.

Tk and Python

- To use the GUI classes in Tk, Python provides us with an interface to Tk in the module called tkinter, short for Tk interface.
- When we work with a graphical user element, we use Python to call the methods of the corresponding graphical object, and tkinter's job is to translate our Python calls into Tk syntax.
- tkinter comes with the Python package so there's no installation needed, but it is not part of the Python core so we need to import it into our code.
- Since tkinter is a long name, a common way to import it is:

```
import tkinter as tk
```

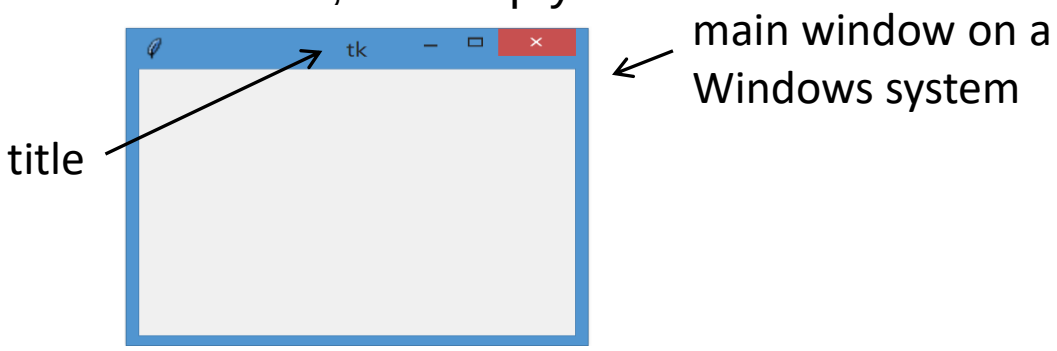
Overview of GUI Functionality

- The GUI for an application is typically put in its own module, with at least one graphical object to provide the user interaction with the application.
- This one object is the main window or root window because it has the main loop method that starts the GUI and keeps the GUI running.
- When the main window is closed, the main loop method stops and the GUI terminates.
- When an application starts up, it instantiates the main window object and calls the mainloop method of the object.
- The mainloop method runs and drives all user interaction with the application, until the X is clicked on the main window object to close it.
- The main window is an object of the Tk class.

```
import tkinter as tk
win = tk.Tk()                  # create the main window object, which is
                               # an object of the Tk class
win.mainloop()                 # run main loop method so that the main
                               # window appears and stays open until X
                               # is clicked
```

Window Title and Background Color

- When a window is first created, it is empty.



- This example window runs on a Windows system so it has the look of a Windows application.
  - The Tk object works with the system window manager so that the icons to minimize, maximize and close the window are already present.
- The default title for the main window is tk, which is the module name.
- To change the title, use the title method: `title('string for title of window')`
- To change the background color of the window, use the bg argument of the configure method: `configure(bg = 'color')`
- Some common Tk colors: black, blue, cyan, gold, gray, green, magenta, maroon, orange, pink, purple, red, tan, white.
- Or see this complete list of colors

Window Size

- By default the window appears at a default minimum size.
- To change the minimum size to a larger dimension, use the method minsize

  ```
  minsize(width, height)
  ```

  where width and height are integers, in pixels.
- By default the window can be resized in both the horizontal and vertical directions.
- To change how the window can be resized, use the resizable method:

  ```
  resizable(horizontal_boolean, vertical_boolean)
  ```

  where the boolean value True means resizable, False means not resizable.
- To change the maximum size that the window can be resized, use the method maxsize

  ```
  maxsize(width, height)
  ```

- To set the size and location of the window:

  ```
  geometry("widthxheight+xStart+yStart")
  ```

  or

  ```
  geometry("+xStart+yStart")
  ```

  - widthxheight are the dimensions of the window, in pixels.
  - xStart+yStart (in pixels) are the (x,y) coordinates of the upper left corner of the window. (0,0) is the upper left corner of the screen.
  - The entire input argument is a string containing no space.

Window Organization

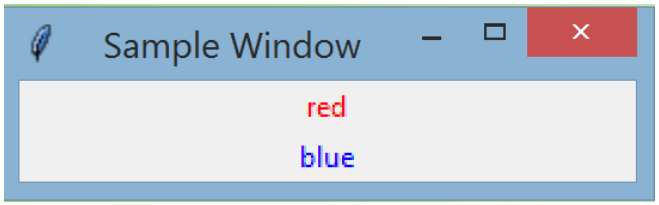
- Once a window is created and optionally set up with a title, size, and location, then we add graphical components such as labels, buttons, etc. to the window.
- Often a window contains multiple components, and we use either the pack or the grid method to configure the location of the components within the window.
  - pack is simple to use but has less flexibility
  - grid is more involved but gives us the ability to easily align components
- Only one method, pack or grid, can be used in a frame or a window.
- To show how pack and grid work to organize graphical components in a window, we first need to create a graphical component. We will use a label component as an example. A label displays a text string in a window.
- To create a label with the text string "red" that's also colored red:

```
L = tk.Label(parentWin, text="red", fg="red")       # create a Label object named L
                # that belongs in a parent window
                                # text string is "red"
                                                # foreground color or text color is red
```

pack

- To use pack, we create the components and pack them together:

```
win = tk.Tk()
win.title("Sample Window")
L = tk.Label(win, text="red", fg="red")
L.pack()
L = tk.Label(win, text="blue", fg="blue")
L.pack()
win.mainloop()
```

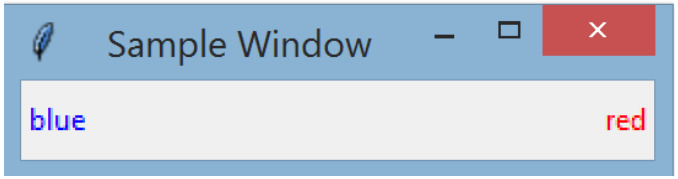


- The default packing direction is vertical, resulting in the above window.
- To change the packing to the horizontal direction, use:

`pack(side=tk.RIGHT)` or `pack(side=tk.LEFT)`

- Example:

```
L = tk.Label(win, text="red", fg="red")
L.pack(side = tk.RIGHT)
L = tk.Label(win, text="blue", fg="blue")
L.pack(side = tk.LEFT)
```

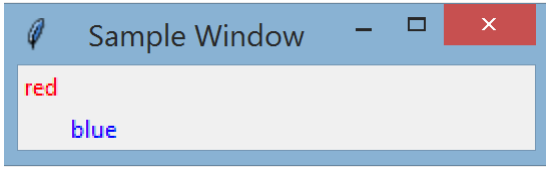


- There are other methods for pack, but for this class we will be using grid.

grid: Row and Column

- grid views the window as a table of rows and columns of cells.
  Row 0 and column 0 is the cell at the upper left corner of the window.
- Example:

```
L1 = tk.Label(win, text="red", fg="red")          # create L1 red label
L2 = tk.Label(win, text="blue", fg="blue")        # create L2 blue label
L1.grid(row=0, column=0)                  # red label is in upper, left corner cell
L2.grid(row=1, column=1)                  # blue label is next row down, next col over
```



- If no row or column is specified, the row defaults to the next available row and the column defaults to 0.
- If a specified row (or column) number is larger than the current max row (or column) number, grid puts the component in the next available row (or column), but this means more components can be inserted in front of the newly specified location.
- If two components are put in the same cell, then the last one that is put in overwrites the first one, and the user sees the last one.

grid: Cell Size

- A row height (or a column width) is dictated by the largest size component that is in the row (or column).
- Sometime a component takes up more space than other components, which means it will make its row (or column) larger than other rows and columns.
- Rather than letting a large component change the size of its row or column, we can tell grid to make the component span multiple rows or columns:

```
grid(row=n, column=m, columnspan = 2)     #  take up columns m and m+1
```

```
grid(row=n, column=m, rowspan = 3)        # take up rows n, n+1, and n+2
```

- The 4 sides of a cell are labeled: n (north or top), s (south or bottom), e (east or right), w (west or left).
- Sometime a small sized component is in a row (or column) that is much larger than its height (or width). By default it will be centered in the cell. But we can set it to be on one side of the cell with the sticky argument:

```
grid(row=n, column=m, sticky='w')     # put component on left side of cell
```

- We can also control the padding (the spacing) around the component:

```
grid(row=n, column=m, padx=N, pady=M)   # N and M are integers that specify
                                        # the padding on the x and y axes
```

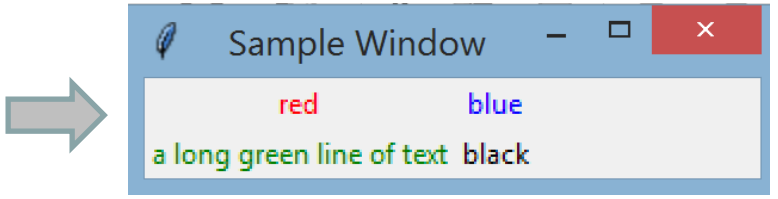
Example of Row, Column, Size

- We have the following 4 colored labels:

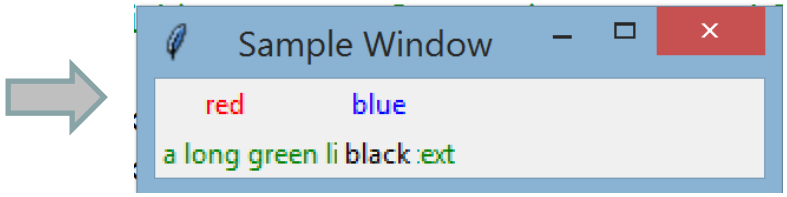
```
L1 = tk.Label(win, text="red", fg="red")
L2 = tk.Label(win, text="blue", fg="blue")
L3 = tk.Label(win, text="a long green line of text", fg="green”)
L4 = tk.Label(win, text="black", fg="black")
```

- We use the following ways to configure their location, with resulting windows:

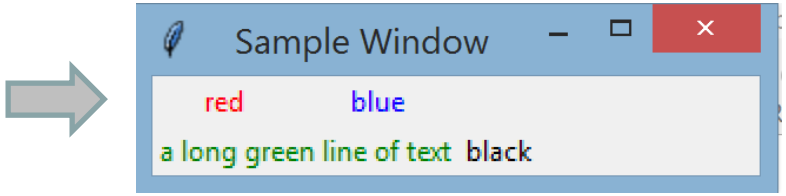
```
L1.grid()
L2.grid(row=0,column=1)
L3.grid(column=0)
L4.grid(row=1,column=1)
```




```
L1.grid()
L2.grid(row=0,column=1)
L3.grid(column=0, columnspan=2)
L4.grid(row=1,column=1)
```




```
L1.grid()
L2.grid(row=0,column=1)
L3.grid(column=0, columnspan=2)
L4.grid(row=1,column=2)
```



grid Method: Resize

- When the user has the option to resize a window, we usually want the cells in the window to also resize:

```
grid_columnconfigure(col_num, weight=n)
```

```
grid_rowconfigure(row_num, weight=n)
```

where:

  - col_num or row_num identifies the column or row that we want to resize.
  - n is a number starting from 1. An n value of 0 is the default weight and means no resize of the row or column.
  - If only one row or one column needs to be resized, then n is 1.
  - If 2 or more rows or columns need to be resized, then the row or column with the higher weight will expand or contract at a faster pace than the other rows or columns. To have 2 or more columns resized at the same pace, use the same value for n.

- The sticky argument is used to expand the size of the component in the cell when a cell is resized:

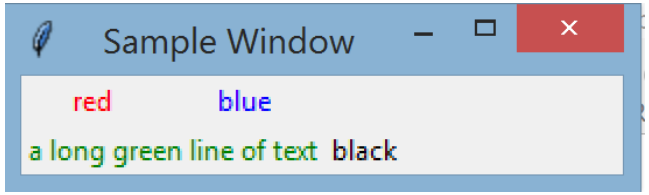
```
grid(row=n, column=m, sticky='we')    # expand the component size
                                      # left-right to match cell resize
```

```
grid(row=n, column=m, sticky='ns')    # expand the component size
                                      # top-bottom to match cell resize
```

Example of Resize

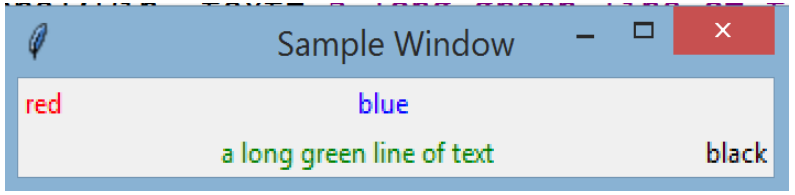
- We have the same 4 colored labels and their locations in the default window size:

```
L1.grid()
L2.grid(row=0,column=1)
L3.grid(column=0, columnspan=2)
L4.grid(row=1,column=2)
```

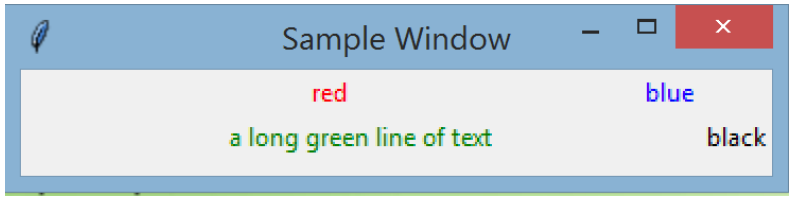


- The following ways resize some of the labels in an expanded windows:

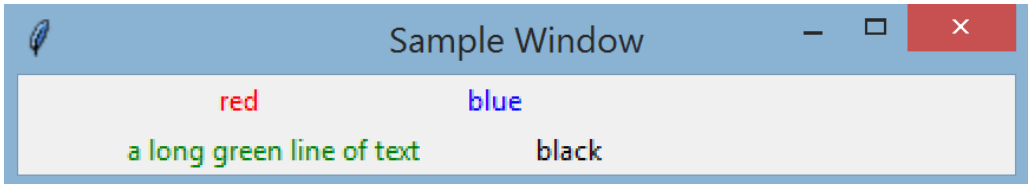
```
# adding to the original code above :
win.grid_columnconfigure(1, weight=1)
      # column 1 expands with window
```




```
# adding to the original code above :
win.grid_columnconfigure(0, weight=1)
      # column 0 expands with window
```



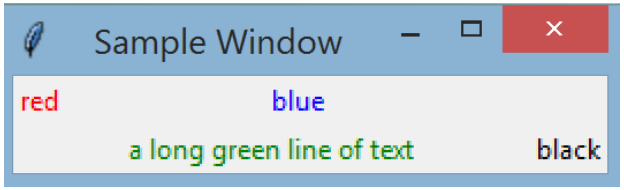

```
# adding to the original code above:
win.grid_columnconfigure(0, weight=1)     # column 0 expands at slower rate
win.grid_columnconfigure(3, weight=2)     # column 2 expands at faster rate
```



Example of Sticky

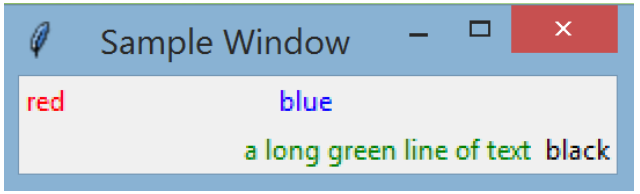
- We have the same 4 colored labels and their locations in the expanded window:

```
L1.grid()
L2.grid(row=0,column=1)
L3.grid(column=0, columnspan=2)
L4.grid(row=1,column=2)
win.grid_columnconfigure(1, weight=1)
```

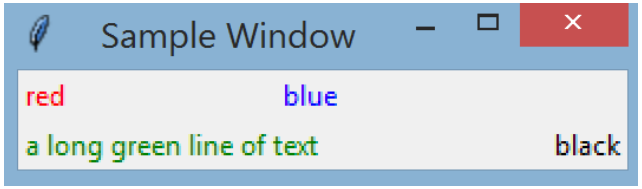


- By default the component is centered in a column or row, but we can set the sticky argument to put it in the ‘n’, ‘s’, ‘e’, ‘w’ position:

```
# changing L3 in the original code above :
L3.grid(column=0, columnspan=2, sticky='e')
```




```
# changing L3 in the original code above :
L3.grid(column=0, columnspan=2, sticky='w')
```



tkinter Data Types

- Often we need to pass a Python data type to a Tk graphical component, which uses Tk data types.
- tkinter provides 4 data classes that we can use to translate our Python data type into a corresponding Tk data type:
  - StringVar
  - IntVar
  - DoubleVar
  - BooleanVar
- The 4 tkinter data classes have:
  - get method: to fetch data out of the data object into a Python data type
  - set method: to store a Python data type into the data object
- Example:

```
myStr = tk.StringVar()          # create tkinter string object
myStr.set("hello")              # store Python string "hello" in tkinter string object,
                                # which translates it to a Tk string
L = tk.Label(textvariable = myStr)      # create a label with the Tk string in myStr
```

Individual Widgets

- The components of the GUI window that display data or get input from the user are called widgets.
- Each widget does a specialized task, and multiple widgets in a GUI work together to give the GUI the functionality that the user experiences.
- There are many Tk widget classes in Tkinter. A comprehensive list of the classes and their attributes are here.
- For this class we work with these individual widgets:
  - Label: display text or image
  - Entry: read in a line of text from the user
  - Button: provide a button for the user to click
  - Radiobutton: provide a radio button for the user to select a choice
  - Canvas: display graphics
  - Listbox: display multiple lines of text that can be selected
  - Scrollbar: used when there are more items than the listbox display size
  - MessageBox: display a pop up message in a new window

Container Widgets

- We also use these container widgets:
  - Frame: a grouping of related widgets
  - Toplevel: a separate window that is spawned from the main window
- Individual widgets are put in a container widget, either a window or a frame. The window or the frame that contains a widget is the master of the widget.

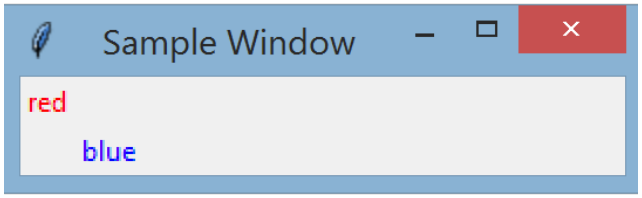
Label Widget

- Display a line of text or an image in a window.
- To create a label:

```
L = tk.Label(master, text="text string to display")      # literal string
L = tk.Label(master, textvariable=aStringVar)            # string in a variable
```

- The first argument, master, is the name of the window or frame object that contains the L widget.
- Recall that when a label contains a StringVar, we can fetch the Python text string from the StringVar object with the method get, and we can store a Python string into the StringVar object with the method set.
- Examples of 2 labels from a previous slide.

```
L1 = tk.Label(win, text="red", fg="red")
L2 = tk.Label(win, text="blue", fg="blue")
L1.grid(row=0, column=0)
L2.grid(row=1, column=1)
```



- If the label contains the text argument, we can change the text string by using: `L['text'] = "new text string to display"`

Text Style in Widgets

- For widgets that contain a text string, we can choose the font type, style, and color.
- The default color is black, but we can change the color:

```
fg="color"          # set foreground or text color
bg="color"          # set background color
```

  where the color are the Tk colors shown in slide 5.

- To change the font type we use a tuple: (font name, height, style) where:
  - The name and style are strings, and the height is an integer
  - We can specify the name only, or name and height only
  - The style can be multiple words in a string
- Examples:

```
font=("Helvetica", 12)          # specify name and height only
font=("Calibri", 14, "bold italic")    # specify all 3 fields,
                                       # with 2 styles
```

- Example of a label:

```
L = tk.Label(win, text="Warning", fg="red", font=("Arial", 14, "bold"))
```

Button Widget

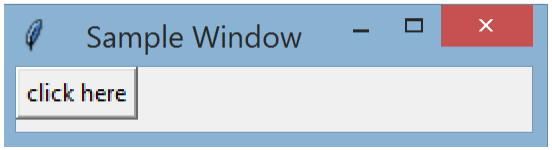
- Display a button for the user to click, which will perform some task.
- To create a button:

```
B = tk.Button(master, text="description", width=N, command=functionName)
```

where
  - text is the description printed on the button
  - width is the size of the button. By default the size of the button is large enough to show all of its text.
  - functionName is a callback function, which will perform a task when the button is pressed.

- Example:

```
def fct() : print("Click!")
B = tk.Button(win, text="click here", command=fct)
B.grid()
```



  - When the user clicks on the button, the button click event causes the callback function fct to run (in other words, the event *calls* the function *back*).
  - The function runs and prints "Click!" to the output window.

Callback Function

- A callback function creates the behavior of the widget when some event (such as a button click) happens to that widget.
- If the callback function has input arguments, we cannot pass the argument when we assign the callback function:

```
val = 6
def printNum(n) :  print(n)
b = tk.Button(text="click here", command=printNum(val))    # doesn't work!
```

  The format: *printNum(val)* is a function call and will cause printNum to run before the button is created!

- Instead we need to use a lambda expression for the function:

```
b = tk.Button(text = "click here", command=lambda : printNum(val))
```

- Now command receives a function reference, not a function call as above.
- In addition, the lambda expression causes a late binding of the printNum function, which means val is not evaluated during compile time but during run time. Therefore val will have its current value when the button is clicked, and not necessarily the value 6 that it was initialized with. This means val can change from one button click to another, depending on factors that modify val.

Radiobutton Widget

- Each radio button is part of a set of multiple radio buttons. Radio buttons let the user select one choice from multiple choices.
- When one button in the set is selected, all the others are automatically unselected.
- To create a radiobutton:

```
RB = tk.Radiobutton(master, text="description", variable=control_var, value=val,
                    command=fct)
```

  - description is a text string to describe the choice for the button
  - control_var is a Tk variable that can store an int or a string. All buttons in the set must use the same control_var.
  - val is the same data type as control_var, it is the unique ID that we give to each radiobutton in the set
  - there can be a callback function for each radio button, but typically radio buttons are to capture a user's choice and not to perform a task.
- After the user has selected a button, the user choice is stored in the control_var.

Example: Radiobutton Widget

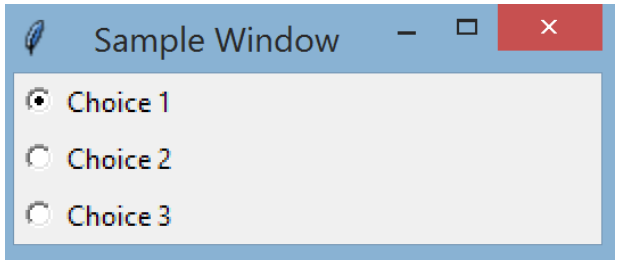
- We create a set of 3 radio buttons:

```
controlVar = tk.IntVar()

rb1 = tk.Radiobutton(win, text="Choice 1", variable=controlVar, value=1)
rb2 = tk.Radiobutton(win, text="Choice 2", variable=controlVar, value=2)
rb3 = tk.Radiobutton(win, text="Choice 3", variable=controlVar, value=3)

rb1.grid()
rb2.grid()
rb3.grid()

controlVar.set(1)
```



- When a choice is clicked, the value of the choice is stored in controlVar. We can write code that uses this value to make a decision to do one task vs. another.

Entry Widget

- Read in a line of text from the user.
- We can refer to characters in the text using indices, which starts at index 0 and goes up until index tk.END
- To create an entry widget

```
E = tk.Entry(master, textvariable=aStringVar)
```

where aStringVar will store the string that the user enters.

- Use get and set methods of the StringVar object to fetch the string or overwrite the string.
- To delete all or part of the string:

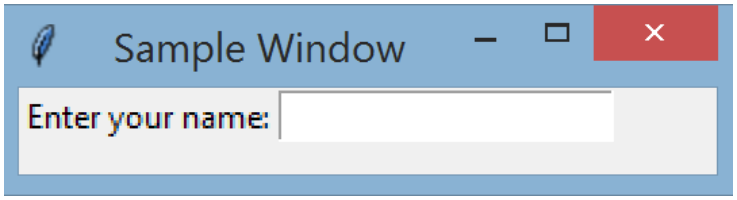
```
E.delete(0, tk.END)      # delete from index 0 to last index, or clear out the string
E.delete(0, last=n)      # delete from index 0 to index n
E.delete(0)              # delete at index 0 only
```

- To add text to the string:

```
E.insert(4, aStringVar)      # aStringVar is inserted at index 4
```

Example: Entry Widget

- The following code shows a label to prompt the user and an entry object for the user to type in a text string.




```
entryText = tk.StringVar()                  # create StringVar to store user input
def fct(event) :                            # callback function that:
    print("Hi,", entryText.get())           #  - prints Hi and user input name
    E.delete(0, tk.END)                     #  - then clears out the entry widget

L = tk.Label(win, text="Enter your name: ")       # create label for prompt
L.grid()

E = tk.Entry(win, textvariable=entryText)   # create entry for user input
E.grid(row=0, column=1)
E.bind("<Return>", fct)                     # bind Return / Enter key to callback fct
```

- In the above code, after the user enters a text string and presses Enter, the Enter / Return key click is the event that causes the callback function to run and print Hi to the user.
- The bind method connects or binds the Return / Enter key click event to the callback function named fct.

bind Method

- The bind method connects or binds an event to a callback function.
- Every widget can bind an event to a callback function:

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

- The event can be a key press (on the keyboard) or a mouse action (such as a mouse click or mouse movement).
- There are many events defined by Tk, the most common is the <Return> event, which is when the user presses the Enter / Return key.
- If a widget binds an event with a callback function: when the event occurs within the widget, then the callback function is called and an event object is passed to the function.
- For a <Return> event we don't need to access the event object, but we still need to write the callback function to accept an event object.
- From the previous example, the callback function accepts an event object even though it doesn't do anything with it:

```
def fct(event) :                    # callback function that:
    print("Hi,", entryText.get())   #  - prints Hi and user input name
    E.delete(0, tk.END)             #  - then clears out the entry widget
```

Canvas Widget (1)

- Display graphical shapes and plots.
- To create a canvas of size m x n, in pixels:

```
C = tk.Canvas(master, width=m, height=n)
```

- The canvas coordinates go from (0,0) at the upper left corner, and increasing x means going right, increasing y means going down.
- To draw a line from (x1,y1) to (x2,y2) coordinates:

```
C.create_line(x1, y1, x2, y2, fill="color", width=n)
```

  where the default fill color is black and the default width is 1.
- To draw a rectangle with (x1,y1) as the upper left corner and (x2, y2) as the lower right corner:

```
C.create_rectangle(x1, y1, x2, y2, fill="color", outline="color", width=n)
```

  where the default fill is no fill, default outline color is black, default width is 1.
- Similar to drawing a rectangle is to draw an oval, which fits within a rectangle with diagonally opposite corners at (x1, y1) and (x2, y2):

```
C.create_oval(x1, y1, x2, y2, fill="color", outline="color", width=n)
```

  where the default fill is no fill, default outline color is black, default width is 1.

Canvas Widget (2)

- The canvas widget can also work with matplotlib to display a plot.
- First we need to set up the connection between Tkinter and matplotlib:

```
import matplotlib
matplotlib.use('TkAgg')                    # tell matplotlib to work with Tkinter
import tkinter as tk                       # normal import of tkinter for GUI
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg  # Canvas widget
import matplotlib.pyplot as plt            # normal import of pyplot to plot
```

- The Canvas widget FigureCanvasTkAgg is a tkinter canvas that's specifically for drawing matplotlib plots.
- The reference to 'backend' is matplotlib's terminology for its interface with various GUI libraries. Here we use the tkinter back end.
- To plot with FigureCanvasTkAgg:

```
fig = plt.figure(figsize=(w, h))           # create a matplotlib figure.  w, h  are the
                                           # width and height of the plot size
# functions to set up the plot go here, such as plt.title, plt.plot…  but not plt.show
canvas = FigureCanvasTkAgg(fig, master=master_win)     # create Canvas widget
canvas.get_tk_widget().grid()              # position the canvas in the window
canvas.draw()                              # since we remove the plt.show above, this is
                                           # used show the plot
```

Listbox Widget (1)

- Display a set of lines of text for the user to select one or more choices.
- To create a listbox:

```
LB = tk.Listbox(master, height=n, width=m, selectmode="multiple")
                        # create an empty listbox that can contain n lines of text
                        # selectmode of extended allows multiple selection,
                        # default selectmode is 1 selection
```

- To insert strings into the listbox:

```
LB.insert(tk.END, aString)        # append aString to the listbox
                                  # typically done in a loop
```

```
LB.insert(tk.END, *aList)         # insert multiple lines in aList
```

- To bind the user click to a callback function:

```
LB.bind('<<ListboxSelect>>', callbackFct)    # the event is triggered when the
                                             # user clicks on a line in the listbox
```

Listbox Widget (2)

- To get the choice(s) that the user selected

```
LB.curselection()        # returns a tuple of the indices matching the selected
                         # strings in the listbox. Index starts at 0 for first string.
```

- To clear the listbox:

```
LB.delete(0, tk.END)     # delete characters from position 0 to the end
```

- To clear the listbox selection:

```
LB.selection_clear(0, tk.END)     # delete all selections
                                  # (from position 0 to last position in the list)
```

- To get all items in the listbox:

```
LB.get(0, tk.END)        # get list items from position 0 to the end of list
```

Scrollbar Widget

- Display a scrollbar for a Listbox widget.
- To create a scrollbar widget

```
S = tk.Scrollbar(master)
```

- Typically a vertical scrollbar is in the same row as the listbox, with ‘ns’ sticky. A horizontal scrollbar is in the same column as the listbox, with ‘we’ sticky.
- To associate the scrollbar with a listbox:
  - Start at the constructor of the listbox

```
L = tk.Listbox(master, height=n, yscrollcommand=S.set)
```

use yscrollcommand for vertical scrollbar,
xscrollcommand for horizontal scrollbar

  - Set the callback for the listbox view function to change when the scrollbar is moved

```
S.config(command=L.yview)
```

use yview for vertical scrollbar
xview for horizontal scrollbar

MessageBox Widget

- Pop up a separate window (a message box) for notification, warning, or error.
- To create a message box, first we need to import the messageBox submodule from TkInter

```
import  tkinter.messagebox  as  tkmb
```

- The messageBox submodule has different types of message boxes.
- We choose a particular type of message box by the method that we use. Here we have 3 common ones:
  - Notification:

```
tkmb.showinfo("title", "notification string", parent=master)
```

  - Error:

```
tkmb.showerror("title", "error string", parent=master)
```

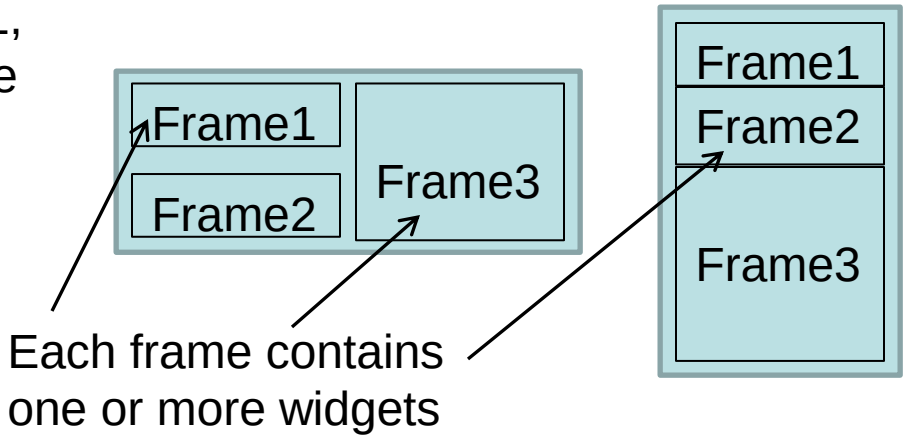
  - Confirmation:

```
tkmb.askokcancel("title", "notification string", parent=master)
```

- The parent=master input argument makes the parent inaccessible to the user until the user clicks to acknowledge the message box.
- For the notification and error messages, the user needs to click OK to acknowledge, then execution continues.
- For the confirmation message, the user can click OK (return value is True) or Cancel (return value is False).
- For the error message, the user can click OK or X (return value is 'ok').
- There are other messageboxes in the messagebox module of Tkinter.

Frame Widget

- Group individual widgets that work together, such as a label and an entry object, into one widget unit.
- This widget unit, or the frame, can then be used to organize groups of related widgets.
- A window can contain multiple frames. And a frame can contain other frames.
- An example of how Frame1, Frame2 and Frame3 can be positioned in 2 different windows:



- In a window that has multiple frames, each frame's layout can be controlled by grid or by pack, independently of the other frames.
- To create a Frame widget:

```
F = tk.Frame(master)
```

- The master of the frame F is the window or frame that contains F. In turn, F is the master for all the widgets that are stored inside it.

Example: Frame Widget

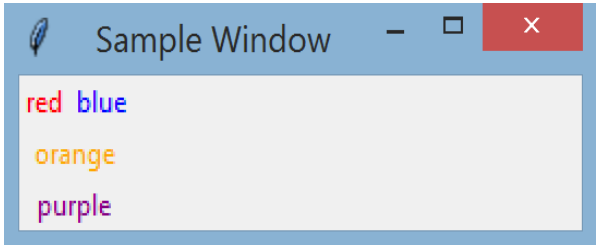
- We create 2 frames F1 and F2.
- F1 uses grid for widget layout, F2 uses pack for widget layout.

```
F1 = tk.Frame(win)
tk.Label(F1, text="red", fg="red").grid()
tk.Label(F1, text="blue", fg="blue").grid(row=0, column=1)

F2 = tk.Frame(win)
tk.Label(F2, text="orange", fg="orange").pack()
tk.Label(F2, text="purple", fg="purple").pack()
```

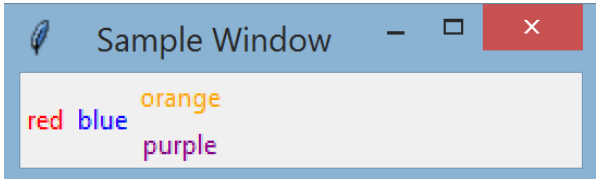
- Placing the 2 frames vertically in a window:

```
F1.grid()
F2.grid(column=0))
```



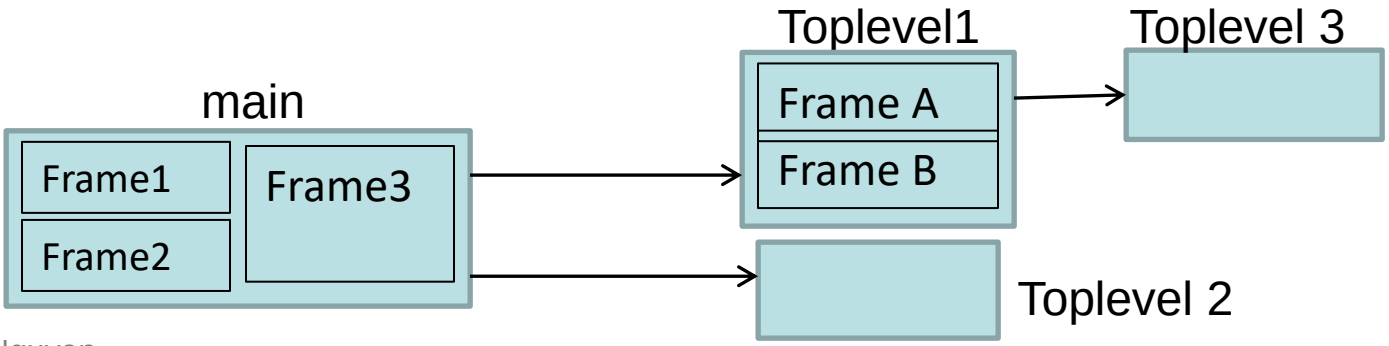
- Placing the 2 frames horizontally in a window:

```
F1.grid()
F2.grid(row=0, column=1)
```



Toplevel Widget

- An additional, separate window from the GUI's main window.
- The top level window is a Tkinter Toplevel object, whereas the main window is a Tkinter Tk object.
- A top level window is created from an existing window. This existing window is the master of the top level window.
- When we create a top level window, we pass to it the reference to the master window.
- To create a Toplevel window: `topWin = tk.Toplevel(master)`
- Once instantiated, the top level window works just like the main window. We can set the title and color, and use grid to add and organize the layout of widgets.
- In the following diagram the main window is the master of Toplevel1 and Toplevel2, and Toplevel1 is the master of Toplevel3.



Coordinate Windows

- With the introduction of top level windows, the GUI will have at least 2 windows that appear on the screen and we need to coordinate them.
- When a top level window appears it can be important to:
  - Disable events in other windows
  - Put the focus on the current top level window
- This way the user input (an event) will come from the source that we expect, and not due to the user randomly clicking on other windows / widgets.
- To do this, in the code to configure the top level window:

```
win.grab_set()        # 'grab' event input, disabling events for other windows
win.focus_set()       # set focus on current window
```

- We also want the top level window to be transient to its master window, or to act like a part of the master, so that the dialog window causes no extra icon on the taskbar:

```
win.transient(master)
```

- If the master of a top level window needs to wait for the top level window to close, before the master resumes its tasks. In the master window, *after creating* the top level window, we add:

```
master.wait_window(topLevel_win)
```

  Now the master will wait until the top level window closes before the master continues to run.

Dialog Window

- One example of a Toplevel window is a dialog window, which is used to get input from the user.
- Some of the widgets in a dialog window can be:
  - A label to prompt the user and an entry object for the user to enter text
  - A set of radio buttons for the user to make a choice
  - Some buttons that the user can click to do some task
  - A listbox that the user can select one or more choices
- The dialog window's purpose is to get the user input and passes the input back to its master window. This means the dialog window should not act on the user's input (such as run some tasks based on the input). Running a task based on the user's input is the job of the master of the dialog window.
- Therefore, when a master window uses the dialog window to get user input, the following steps are needed:
  - The master window starts the dialog window
  - The master window pauses to wait for the dialog window to close
  - The dialog window interacts with the user to get the user input
  - When the dialog window has the user input, it closes itself
  - The master window runs again, gets the user input from the dialog window, and processes the user input

Closing Windows (1)

- When a window simply displays some data to the user, and then the user clicks X to close the window, then the system window manager takes care of closing the window and putting the focus back on the master.
- But at times we don't want the user to have to click X to close the window. We want our code to close the current window. To close the current window:

```
win.destroy()          # close current window named win
```

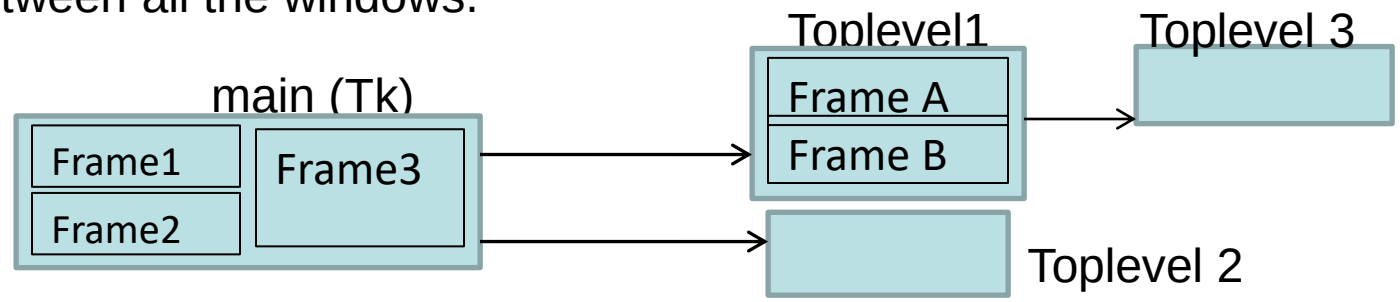
- Sometime when the user clicks X to close the current window, we may not want the window to close immediately if some process is running.
  - Instead we want to stop the process, close any resources that are being used, and then close the window.
- To override the default event handling of the system window manager when the user clicks X, we use:

```
win.protocol("WM_DELETE_WINDOW", callback_fct)
```

  - protocol interacts with the system window manager
  - WM_DELETE_WINDOW is the window manager's name for the event of the X being clicked
  - The callback function is the code we provide that will do all the tasks we want when the user clicks X.

Closing Windows (2)

- A GUI application is designed correctly if there is one main window (Tk window) and a number of Toplevel windows, and the Toplevel windows' master is the main window or another Toplevel window. There is a 'link' between all the windows.



- When the main window is closed, the close signal follows all the links to close all the Toplevel windows also.
- But closing a window does not mean the window object is gone. The object stays in memory as long as there is still a reference to it.
  For example, if the Toplevel1 window is closed, the Toplevel1 object is still in memory because the main window still has a reference to it.
- Likewise, if the main window is closed, all the windows will be closed, but the window objects are still in memory, and the mainloop() still runs. To end the mainloop and end the application, when the main window closes, it should also run:

```
win.quit()
```

Event Driven Programming

- GUI programming uses event driven programming, which is a new way of looking at how the code runs. This is similar to how *object oriented programming* is a new way of looking at how code runs compared to *procedural programming* (with functions).
- The GUI starts and ends with the mainloop method of the main window.

```
win = tk.Tk()                  # begin of GUI code, create main window
win.mainloop()                 # when the main window first appears
```

- The mainloop is an infinite loop that keeps running to process *events* until one specific *event* happens, when the X is clicked on the main window. Our code cannot "return" from the main window to end the GUI, unlike the return statement from a main function.
- Likewise, our GUI code consists of:
  - Creating widgets, customizing them, placing them in the proper place
  - Writing callback functions to do work and associating these functions to a widget
- But our code doesn't directly call the callback functions! Instead, these functions are "called back" or run only because an event occurred.

Structuring GUI Code (1)

- When implementing a complete GUI for an application:
  - We may need multiple windows, each with methods to set up the window.
  - Each window can have multiple frames, each with methods to set up each frame.
  - Each frame can have multiple widgets, each with set up methods and callback functions.
- That's a lot of code for the GUI components to interact with each other, but the code does not follow a classic flow chart that goes from start to end because this is event driven programming! The user can choose to jump around the different components, causing different methods to run.
- The best way to organize all the GUI components and their respective methods is to use OOP.
- Each window is written as a class which is *derived from the tkinter class*.
- This way the window class inherits all the tkinter functionalities, and the class has the following methods:
  - The constructor that has:
    - All the methods to set up the window
    - All widgets and their configuration
  - The callback functions for all the widgets

Structuring GUI Code (2)

- Example outline of a top level window class:

```
class myWin(tk.Toplevel) :              # inherit from tkinter Toplevel class
    def __init__(self, master) :
        super().__init__(master)
        # code to set up the window here. For example:
        self.title("sample top level window")
        # code to configure widgets here. For example:
        tk.Button(self, text="Click here", command=self.method1).grid()
    # other methods here. For example:
    def method1(self) :
        print("Clicked!")
```

- The main window class has the same organization as a top level window, except there's no master. Example outline of a main window class:

```
class myWin(tk.Tk) :                    # inherit from tkinter Tk class
    def __init__(self) :
        super().__init__()
        # the rest code to set up the window here…
```

Going Further…

- We've had a brief introduction to the world of GUI programming.
- More information such as other widgets, different styling of widgets, etc. can be found in the New Mexico Tech Computer Center documentation.
- tkinter is the "classic" GUI module for Python but there are also other popular GUI packages that provide even more functionalities.
  - Most notable are wxPython and pyQt. Both of these are written in C++, which makes it convenient to enhance a feature with our own C++ code.

Up next: Web Access