Chapter 16

GUI Programming II

In this chapter we cover more basic GUI concepts.

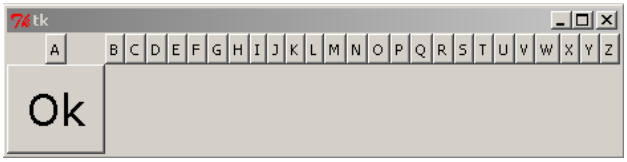
16.1 Frames

Let's say we want 26 small buttons across the top of the screen, and a big Ok button below them, like below:

We try the following code:

```
from tkinter import *

root = Tk()

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
buttons = [0]*26
for i in range(26):
    buttons[i] = Button(text=alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))
ok_button.grid(row=1, column=0)

mainloop()
```

But we instead get the following unfortunate result:



The problem is with column 0. There are two widgets there, the A button and the Ok button, and Tkinter will make that column big enough to handle the larger widget, the Ok button. One solution to this problem is shown below:

```
ok_button.grid(row=1, column=0, columnspan=26)
```

Another solution to this problem is to use what is called a *frame*. The frame's job is to hold other widgets and essentially combine them into one large widget. In this case, we will create a frame to group all of the letter buttons into one large widget. The code is shown below:

```
from tkinter import *

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
root = Tk()

button_frame = Frame()
buttons = [0]*26
for i in range(26):
    buttons[i] = Button(button_frame, text=alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))

button_frame.grid(row=0, column=0)
ok_button.grid(row=1, column=0)

mainloop()
```

To create a frame, we use `Frame()` and give it a name. Then, for any widgets we want include in the frame, we include the name of the frame as the first argument in the widget's declaration. We still have to grid the widgets, but now the rows and columns will be relative to the frame. Finally, we have to grid the frame itself.

16.2 Colors

Tkinter defines many common color names, like `'yellow'` and `'red'`. It also provides a way to get access to millions of more colors. We first have to understand how colors are displayed on the screen.

Each color is broken into three components—a red, a green, and a blue component. Each component can have a value from 0 to 255, with 255 being the full amount of that color. Equal parts of red and green create shades of yellow, equal parts of red and blue create shades of purple, and equal

parts of blue and green create shades of turquoise. Equal parts of all three create shades of gray. Black is when all three components have values of 0 and white is when all three components have values of 255. Varying the values of the components can produce up to $256^3 \approx 16$ million colors. There are a number of resources on the web that allow you to vary the amounts of the components and see what color is produced.

To use colors in Tkinter is easy, but with one catch—component values are given in hexadecimal. Hexadecimal is a base 16 number system, where the letters A-F are used to represent the digits 10 through 15. It was widely used in the early days of computing, and it is still used here and there. Here is a table comparing the two number bases:

0	0	8	8	16	10	80	50
1	1	9	9	17	11	100	64
2	2	10	A	18	12	128	80
3	3	11	B	31	1F	160	A0
4	4	12	C	32	20	200	C8
5	5	13	D	33	21	254	FE
6	6	14	E	48	30	255	FF
7	7	15	F	64	40	256	100

Because the color component values run from 0 to 255, they will run from 0 to FF in hexadecimal, and thus are described by two hex digits. A typical color in Tkinter is specified like this: `'#A202FF'`. The color name is prefaced with a pound sign. Then the first two digits are the red component (in this case `A2`, which is 162 in decimal). The next two digits specify the green component (here `02`, which is 2 in decimal), and the last two digits specify the blue component (here `FF`, which is 255 in decimal). This color turns out to be a bluish violet. Here is an example of it in use:

```
label = Label(text='Hi', bg='#A202FF')
```

If you would rather not bother with hexadecimal, you can use the following function which will convert percentages into the hex string that Tkinter uses.

```
def color_convert(r, g, b):
    return '#{:02x}{:02x}{:02x}'.format(int(r*2.55),int(g*2.55),
                                        int(b*2.55))
```

Here is an example of it to create a background color that has 100% of the red component, 85% of green and 80% of blue.

```
label = Label(text='Hi', bg=color_convert(100, 85, 80))
```

16.3 Images

Labels and buttons (and other widgets) can display images instead of text.

To use an image requires a little set-up work. We first have to create a `PhotoImage` object and give it a name. Here is an example:

```
cheetah_image = PhotoImage(file='cheetahs.gif')
```

Here are some examples of putting the image into widgets:

```
label = Label(image=cheetah_image)
button = Button(image=cheetah_image, command=cheetah_callback())
```

You can use the `configure` method to set or change an image:

```
label.configure(image=cheetah_image)
```

File types One unfortunate limitation of Tkinter is the only common image file type it can use is GIF. If you would like to use other types of files, one solution is to use the Python Imaging Library, which will be covered in Section 18.2.

16.4 Canvases

A canvas is a widget on which you can draw things like lines, circles, rectangles. You can also draw text, images, and other widgets on it. It is a very versatile widget, though we will only describe the basics here.

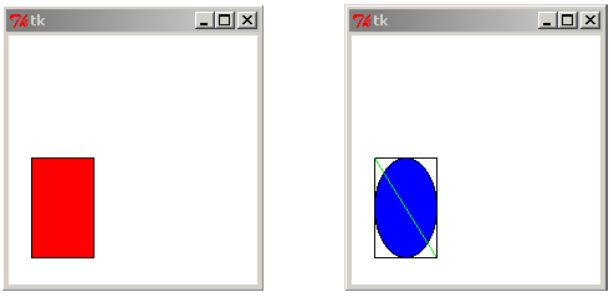
Creating canvases The following line creates a canvas with a white background that is 200 × 200 pixels in size:

```
canvas = Canvas(width=200, height=200, bg='white')
```

Rectangles The following code draws a red rectangle to the canvas:

```
canvas.create_rectangle(20,100,30,150, fill='red')
```

See the image below on the left. The first four arguments specify the coordinates of where to place the rectangle on the canvas. The upper left corner of the canvas is the origin, $(0, 0)$. The upper left of the rectangle is at $(20, 100)$, and the lower right is at $(30, 150)$. If were to leave off `fill='red'`, the result would be a rectangle with a black outline.



Ovals and lines Drawing ovals and lines is similar. The image above on the right is created with the following code:

```
canvas.create_rectangle(20,100,70,180)
canvas.create_oval(20,100,70,180, fill='blue')
canvas.create_line(20,100,70,180, fill='green')
```

The rectangle is here to show that lines and ovals work similarly to rectangles. The first two coordinates are the upper left and the second two are the lower right.

To get a circle with radius `r` and center `(x,y)`, we can create the following function:

```
def create_circle(x,y,r):
    canvas.create_oval(x-r,y-r,x+r,y+r)
```

Images We can add images to a canvas. Here is an example:

```
cheetah_image = PhotoImage(file='cheetahs.gif')
canvas.create_image(50,50, image=cheetah_image)
```

The two coordinates are where the center of the image should be.

Naming things, changing them, moving them, and deleting them We can give names to the things we put on the canvas. We can then use the name to refer to the object in case we want to move it or remove it from the canvas. Here is an example were we create a rectangle, change its color, move it, and then delete it:

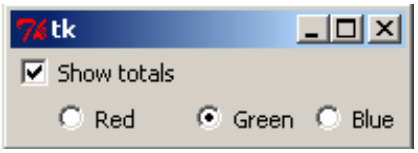
```
rect = canvas.create_rectangle(0,0,20,20)
canvas.itemconfigure(rect, fill='red')
canvas.coords(rect,40,40,60,60)
canvas.delete(rect)
```

The `coords` method is used to move or resize an object and the `delete` method is used to delete it. If you want to delete everything from the canvas, use the following:

```
canvas.delete(ALL)
```

16.5 Check buttons and Radio buttons

In the image below, the top line shows a check button and the bottom line shows a radio button.



Check buttons The code for the above check button is:

```
show_totals = IntVar()
check = Checkbutton(text='Show totals', var=show_totals)
```

The one thing to note here is that we have to tie the check button to a variable, and it can't be just any variable, it has to be a special kind of Tkinter variable, called an `IntVar`. This variable, `show_totals`, will be 0 when the check button is unchecked and 1 when it is checked. To access the value of the variable, you need to use its `get` method, like this:

```
show_totals.get()
```

You can also set the value of the variable using its `set` method. This will automatically check or uncheck the check button on the screen. For instance, if you want the above check button checked at the start of the program, do the following:

```
show_totals = IntVar()
show_totals.set(1)
check = Checkbutton(text='Show totals', var=show_totals)
```

Radio buttons Radio buttons work similarly. The code for the radio buttons shown at the start of the section is:

```
color = IntVar()
redbutton = Radiobutton(text='Red', var=color, value=1)
greenbutton = Radiobutton(text='Green', var=color, value=2)
bluebutton = Radiobutton(text='Blue', var=color, value=3)
```

The value of the `IntVar` object `color` will be 1, 2, or 3, depending on whether the left, middle, or right button is selected. These values are controlled by the `value` option, specified when we create the radio buttons.

Commands Both check buttons and radio buttons have a `command` option, where you can set a callback function to run whenever the button is selected or unselected.

16.6 `Text` widget

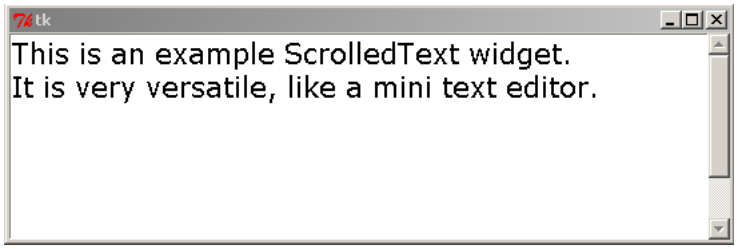
The `Text` widget is a bigger, more powerful version of the `Entry` widget. Here is an example of creating one:

```
textbox = Text(font=('Verdana', 16), height=6, width=40)
```

The widget will be 40 characters wide and 6 rows tall. You can still type past the sixth row; the widget will just display only six rows at a time, and you can use the arrow keys to scroll.

If you want a scrollbar associated with the text box you can use the `ScrolledText` widget. Other than the scrollbar, `ScrolledText` works more or less the same as `Text`. An example is of what it looks like is shown below. To use the `ScrolledText` widget, you will need the following import:

```
from tkinter.scrolledtext import ScrolledText
```



Here are a few common commands:

Statement	Description
`textbox.get(1.0,END)`	returns the contents of the text box
`textbox.delete(1.0,END)`	deletes everything in the text box
`textbox.insert(END,'Hello')`	inserts text at the end of the text box

One nice option when declaring the `Text` widget is `undo=True`, which allows Ctrl+Z and Ctrl+Y to undo and redo edits. There are a ton of other things you can do with the `Text` widget. It is almost like a miniature word processor.

16.7 `Scale` widget

A `Scale` is a widget that you can slide back and forth to select different values. An example is shown below, followed by the code that creates it.


```
scale = Scale(from_=1, to_=100, length=300, orient='horizontal')
```

Here are some of the useful options of the `Scale` widget:

Option	Description
`from_`	minimum value possible by dragging the scale
`to_`	maximum value possible by dragging the scale
`length`	how many pixels long the scale is
`label`	specify a label for the scale
`showvalue='NO'`	gets rid of the number that displays above the scale
`tickinterval=1`	displays tickmarks at every unit (1 can be changed)

There are several ways for your program to interact with the scale. One way is to link it with an `IntVar` just like with check buttons and radio buttons, using the `variable` option. Another option is to use the scale's `get` and `set` methods. A third way is to use the `command` option, which

works just like with buttons.

16.8 GUI Events

Often we will want our programs to do something if the user presses a certain key, drags something on a canvas, uses the mouse wheel, etc. These things are called *events*.

A simple example The first GUI program we looked at back in Section 15.1 was a simple temperature converter. Anytime we wanted to convert a temperature we would type in the temperature in the entry box and click the Calculate button. It would be nice if the user could just press the enter key after they type the temperature instead of having to click to Calculate button. We can accomplish this by adding one line to the program:

```
entry.bind('<Return>', lambda dummy=0:calculate())
```

This line should go right after you declare the entry box. What it does is it takes the event that the enter (return) key is pressed and *binds* it to the `calculate` function.

Well, sort of. The function you bind the event to is supposed to be able to receive a copy of an `Event` object, but the calculate function that we had previously written takes no arguments. Rather than rewrite the function, the line above uses `lambda` trick to essentially throw away the `Event` object.

Common events Here is a list of some common events:

Event	Description
`<Button-1>`	The left mouse button is clicked.
`<Double-Button-1>`	The left mouse button is double-clicked.
`<Button-Release-1>`	The left mouse button is released.
`<B1-Motion>`	A click-and-drag with the left mouse button.
`<MouseWheel>`	The mouse wheel is moved.
`<Motion>`	The mouse is moved.
`<Enter>`	The mouse is now over the widget.
`<Leave>`	The mouse has now left the widget.
`<Key>`	A key is pressed.
<key name>	The *key name* key is pressed.

For all of the mouse button examples, the number 1 can be replaced with other numbers. Button 2 is the middle button and button 3 is the right button.

The most useful attributes in the `Event` object are:

Attribute	Description
`keysym`	The name of the key that was pressed
`x, y`	The coordinates of the mouse pointer
`delta`	The value of the mouse wheel

Key events For key events, you can either have specific callbacks for different keys or catch all keypresses and deal with them in the same callback. Here is an example of the latter:

```
from tkinter import *

def callback(event):
    print(event.keysym)

root = Tk()
root.bind('<Key>', callback)

mainloop()
```

The above program prints out the names of the keys that were pressed. You can use those names in if statements to handle several different keypresses in the callback function, like below:

```
if event.keysym == 'percent':
    # percent (shift+5) was pressed, do something about it...
elif event.keysym == 'a':
    # lowercase a was pressed, do something about it...
```

Use the single callback method if you are catching a lot of keypresses and are doing something similar with all of them. On the other hand, if you just want to catch a couple of specific keypresses or if certain keys have very long and specific callbacks, you can catch keypresses separately like below:

```
from tkinter import *

def callback1(event):
    print('You pressed the enter key.')

def callback2(event):
    print('You pressed the up arrow.')

root = Tk()
root.bind('<Return>', callback1)
root.bind('<Up>', callback2)
mainloop()
```

The key names are the same as the names stored in the keysym attribute. You can use the program from earlier in this section to find the names of all the keys. Here are the names for a few common keys:

Tkinter name	Common name
<Return>	Enter key
<Tab>	Tab key
<Space>	Spacebar
<F1>, ..., <F12>	F1, ..., F12
<Next>, <Prior>	Page up, Page down
<Up>, <Down>, <Left>, <Right>	Arrow keys
<Home>, <End>	Home, End
<Insert>, <Delete>	Insert, Delete
<Caps_Lock>, <Num_Lock>	Caps lock, Number lock
<Control_L>, <Control_R>	Left and right Control keys
<Alt_L>, <Alt_R>	Left and right Alt keys
<Shift_L>, <Shift_R>	Left and right Shift keys

Most printable keys can be captured with their names, like below:

```
root.bind('a', callback)
root.bind('A', callback)
root.bind('-', callback)
```

The exceptions are the spacebar (<Space>) and the less than sign (<Less>). You can also catch key combinations, such as <Shift-F5>, <Control-Next>, <Alt-2>, or <Control-Shift-F1>.

Note These examples all bind keypresses to root, which is our name for the main window. You can also bind keypresses to specific widgets. For instance, if you only want the left arrow key to work on a Canvas called canvas, you could use the following:

```
canvas.bind(<Left>, callback)
```

One trick here, though, is that the canvas won't recognize the keypress unless it has the GUI's focus. This can be done as below:

```
canvas.focus_set()
```

16.9 Event examples

Example 1 Here is an example where the user can move a rectangle with the left or right arrow keys.

```
from tkinter import *

def callback(event):
    global move
    if event.keysym=='Right':
```

```
        move += 1
    elif event.keysym=='Left':
        move -=1
    canvas.coords(rect,50+move,50,100+move,100)

root = Tk()
root.bind('<Key>', callback)
canvas = Canvas(width=200,height=200)
canvas.grid(row=0,column=0)
rect = canvas.create_rectangle(50,50,100,100,fill='blue')
move = 0

mainloop()
```

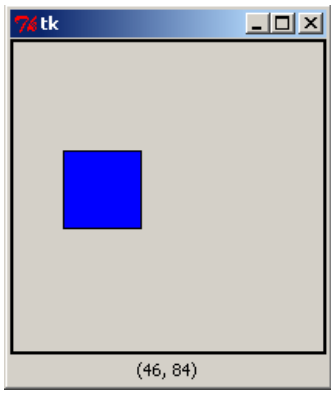
Example 2 Here is an example program demonstrating mouse events. The program starts by drawing a rectangle to the screen. The user can do the following:

- Drag the rectangle with the mouse (`<B1_Motion>`).
- Resize the rectangle with the mouse wheel (`<MouseWheel>`).
- Whenever the user left-clicks, the rectangle will change colors (`<Button-1>`).
- Anytime the mouse is moved, the current coordinates of the mouse are displayed in a label (`<Motion>`).

Here is the code for the program:

```
from tkinter import *

def mouse_motion_event(event):
    label.configure(text='({}, {})'.format(event.x, event.y))

def wheel_event(event):
    global x1, x2, y1, y2
    if event.delta>0:
        diff = 1
    elif event.delta<0:
        diff = -1
    x1+=diff
    x2-=diff
    y1+=diff
    y2-=diff
    canvas.coords(rect,x1,y1,x2,y2)

def b1_event(event):
    global color
    if not b1_drag:
        color = 'Red' if color=='Blue' else 'Blue'
        canvas.itemconfigure(rect, fill=color)
```

```
def b1_motion_event(event):
    global b1_drag, x1, x2, y1, y2, mouse_x, mouse_y
    x = event.x
    y = event.y
    if not b1_drag:
        mouse_x = x
        mouse_y = y
        b1_drag = True
        return
    x1+=(x-mouse_x)
    x2+=(x-mouse_x)
    y1+=(y-mouse_y)
    y2+=(y-mouse_y)
    canvas.coords(rect,x1,y1,x2,y2)
    mouse_x = x
    mouse_y = y

def b1_release_event(event):
    global b1_drag
    b1_drag = False

root=Tk()

label = Label()

canvas = Canvas(width=200, height=200)
canvas.bind('<Motion>', mouse_motion_event)
canvas.bind('<ButtonPress-1>', b1_event)
canvas.bind('<B1-Motion>', b1_motion_event)
canvas.bind('<ButtonRelease-1>', b1_release_event)
canvas.bind('<MouseWheel>', wheel_event)
canvas.focus_set()

canvas.grid(row=0, column=0)
label.grid(row=1, column=0)

mouse_x = 0
mouse_y = 0
b1_drag = False

x1 = y1 = 50
x2 = y2 = 100
color = 'blue'
rect = canvas.create_rectangle(x1,y1,x2,y2,fill=color)

mainloop()
```



Here are a few notes about how the program works:

1. First, every time the mouse is moved over the canvas, the `mouse_motion_event` function is called. This function prints the mouse's current coordinates which are contained in the `Event` attributes `x` and `y`.

2. The `wheel_event` function is called whenever the user uses the mouse (scrolling) wheel. The `Event` attribute `delta` contains information about how quickly and in what direction the wheel was moved. We just stretch or shrink the rectangle based on whether the wheel was moved forward or backward.

3. The `b1_event` function is called whenever the user presses the left mouse button. The function changes the color of the rectangle whenever the rectangle is clicked. There is a global variable here called `b1_drag` that is important. It is set to **`True`** whenever the user is dragging the rectangle. When dragging is going on, the left mouse button is down and the `b1_event` function is continuously being called. We don't want to keep changing the color of the rectangle in that case, hence the if statement.

4. The dragging is accomplished mostly in the `b1_motion_event` function, which is called whenever the left mouse button is down and the mouse is being moved. It uses global variables that keep track of what the mouse's position was the last time the function was called, and then moves the rectangle according to the difference between the new and old position.

   When the dragging is down, the left mouse button will be released. When that happens, the `b1_release_event` function is called, and we set the global `b1_drag` variable accordingly.

5. The `focus_set` method is needed because the canvas will not recognize the mouse wheel events unless the focus is on the canvas.

6. One problem with this program is that the user can modify the rectangle by clicking anywhere on the canvas, not just on rectangle itself. If we only want the changes to happen when the mouse is over the rectangle, we could specifically bind the rectangle instead of the whole canvas, like below:

```
canvas.tag_bind(rect, '<B1-Motion>', b1_motion_event)
```

7. Finally, the use of global variables here is a little messy. If this were part of a larger project, it might make sense to wrap all of this up into a class.