Chapter 15

GUI Programming with Tkinter

Up until now, the only way our programs have been able to interact with the user is through keyboard input via the `input` statement. But most real programs use windows, buttons, scrollbars, and various other things. These *widgets* are part of what is called a *Graphical User Interface* or GUI. This chapter is about GUI programming in Python with Tkinter.

All of the widgets we will be looking at have far more options than we could possibly cover here. An excellent reference is Fredrik Lundh's *Introduction to Tkinter* [2].

15.1 Basics

Nearly every GUI program we will write will contain the following three lines:

```
from tkinter import *
root = Tk()
mainloop()
```

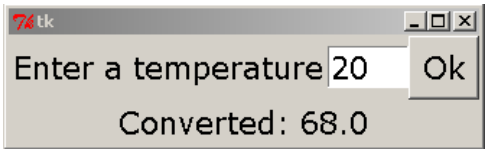
The first line imports all of the GUI stuff from the `tkinter` module. The second line creates a window on the screen, which we call `root`. The third line puts the program into what is essentially a long-running while loop called the *event loop*. This loop runs, waiting for keypresses, button clicks, etc., and it exits when the user closes the window.

Here is a working GUI program that converts temperatures from Fahrenheit to Celsius.

```
from tkinter import *

def calculate():
    temp = int(entry.get())
    temp = 9/5*temp+32
    output_label.configure(text = 'Converted: {:.1f}'.format(temp))
    entry.delete(0,END)
```

```
root = Tk()
message_label = Label(text='Enter a temperature',
                      font=('Verdana', 16))
output_label = Label(font=('Verdana', 16))
entry = Entry(font=('Verdana', 16), width=4)
calc_button = Button(text='Ok', font=('Verdana', 16),
                     command=calculate)

message_label.grid(row=0, column=0)
entry.grid(row=0, column=1)
calc_button.grid(row=0, column=2)
output_label.grid(row=1, column=0, columnspan=3)

mainloop()
```

Here is what the program looks like:



We now will examine the components of the program separately.

15.2 Labels

A label is a place for your program to place some text on the screen. The following code creates a label and places it on the screen.

```
hello_label = Label(text='hello')
hello_label.grid(row=0, column=0)
```

We call `Label` to create a new label. The capital `L` is required. Our label's name is `hello_label`. Once created, use the `grid` method to place the label on the screen. We will explain `grid` in the next section.

Options There are a number of options you can change including font size and color. Here are some examples:

```
hello_label = Label(text='hello', font=('Verdana', 24, 'bold'),
                    bg='blue', fg='white')
```

Note the use of keyword arguments. Here are a few common options:

- `font` — The basic structure is `font=` (*font name, font size, style*). You can leave out the font size or the style. The choices for style are `'bold'`, `'italic'`, `'underline'`, `'overstrike'`, `'roman'`, and `'normal'` (which is the default). You can combine multiple styles like this: `'bold italic'`.

- `fg` and `bg` — These stand for `foreground` and `background`. Many common color names can be used, like `'blue'`, `'green'`, etc. Section 16.2 describes how to get essentially any color.
- `width` — This is how many characters long the label should be. If you leave this out, Tkinter will base the width off of the text you put in the label. This can make for unpredictable results, so it is good to decide ahead of time how long you want your label to be and set the `width` accordingly.
- `height` — This is how many rows high the label should be. You can use this for multi-line labels. Use newline characters in the text to get it to span multiple lines. For example, `text='hi\nthere'`.

There are dozens more options. The aforementioned *Introduction to Tkinter* [2] has a nice list of the others and what they do.

Changing label properties Later in your program, after you've created a label, you may want to change something about it. To do that, use its `configure` method. Here are two examples that change the properties of a label called `label`:

```
label.configure(text='Bye')
label.configure(bg='white', fg='black')
```

Setting `text` to something using the `configure` method is kind of like the GUI equivalent of a **`print`** statement. However, in calls to `configure` we cannot use commas to separate multiple things to print. We instead need to use string formatting. Here is a **`print`** statement and its equivalent using the `configure` method.

```
print('a =', a, 'and b =', b)
label.configure(text='a = {}, and b = {}'.format(a,b))
```

The `configure` method works with most of the other widgets we will see.

15.3 `grid`

The `grid` method is used to place things on the screen. It lays out the screen as a rectangular grid of rows and columns. The first few rows and columns are shown below.

`(row=0, column=0)`	`(row=0, column=1)`	`(row=0, column=2)`
`(row=1, column=0)`	`(row=1, column=1)`	`(row=1, column=2)`
`(row=2, column=0)`	`(row=2, column=1)`	`(row=2, column=2)`

Spanning multiple rows or columns There are optional arguments, `rowspan` and `columnspan`, that allow a widget to take up more than one row or column. Here is an example of several grid statements followed by what the layout will look like:

```
label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=0, columnspan=2)
label4.grid(row=1, column=2)
label5.grid(row=2, column=2)
```

label1	label2	
label 3		label4
		label5

Spacing To add extra space between widgets, there are optional arguments `padx` and `pady`.

Important note Any time you create a widget, to place it on the screen you need to use `grid` (or one of its cousins, like `pack`, which we will talk about later). Otherwise it will not be visible.

15.4 Entry boxes

Entry boxes are a way for your GUI to get text input. The following example creates a simple entry box and places it on the screen.

```
entry = Entry()
entry.grid(row=0, column=0)
```

Most of the same options that work with labels work with entry boxes (and most of the other widgets we will talk about). The `width` option is particularly helpful because the entry box will often be wider than you need.

- **Getting text** To get the text from an entry box, use its `get` method. This will return a string. If you need numerical data, use **`eval`** (or **`int`** or **`float`**) on the string. Here is a simple example that gets text from an entry box named `entry`.

  ```
  string_value = entry.get()
  num_value = eval(entry.get())
  ```

- **Deleting text** To clear an entry box, use the following:

  ```
  entry.delete(0,END)
  ```

- **Inserting text** To insert text into an entry box, use the following:

  ```
  entry.insert(0, 'hello')
  ```

15.5 Buttons

The following example creates a simple button:

```
ok_button = Button(text='Ok')
```

To get the button to do something when clicked, use the `command` argument. It is set to the name of a function, called a *callback function*. When the button is clicked, the callback function is called. Here is an example:

```
from tkinter import *

def callback():
    label.configure(text='Button clicked')

root = Tk()
label = Label(text='Not clicked')
button = Button(text='Click me', command=callback)

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

mainloop()
```

When the program starts, the label says `Click me`. When the button is clicked, the callback function `callback` is called, which changes the label to say `Button clicked`.

`lambda` trick Sometimes we will want to pass information to the callback function, like if we have several buttons that use the same callback function and we want to give the function information about which button is being clicked. Here is an example where we create 26 buttons, one for each letter of the alphabet. Rather than use 26 separate `Button()` statements and 26 different functions, we use a list and one function.

```
from tkinter import *
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def callback(x):
    label.configure(text='Button {} clicked'.format(alphabet[x]))

root = Tk()

label = Label()
label.grid(row=1, column=0, columnspan=26)

buttons = [0]*26 # create a list to hold 26 buttons
for i in range(26):
    buttons[i] = Button(text=alphabet[i],
                        command = lambda x=i: callback(x))
```

```
    buttons[i].grid(row=0, column=i)

mainloop()
```

We note a few things about this program. First, we set `buttons=[0]*26`. This creates a list with 26 things in it. We don't really care what thoset things are because they will be replaced with buttons. An alternate way to create the list would be to set `buttons=[]` and use the `append` method.

We only use one callback function and it has one argument, which indicates which button was clicked. As far as the **lambda** trick goes, without getting into the details, `command=callback(i)` does not work, and that is why we resort to the **lambda** trick. You can read more about **lambda** in Section 23.2. An alternate approach is to use classes.

15.6 Global variables

Let's say we want to keep track of how many times a button is clicked. An easy way to do this is to use a global variable as shown below.

```
from tkinter import *

def callback():
    global num_clicks
    num_clicks = num_clicks + 1
    label.configure(text='Clicked {} times.'.format(num_clicks))

num_clicks = 0
root = Tk()

label = Label(text='Not clicked')
button = Button(text='Click me', command=callback)

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

mainloop()
```

We will be using a few global variables in our GUI programs. Using global variables unnecessarily, especially in long programs, can cause difficult to find errors that make programs hard to maintain,

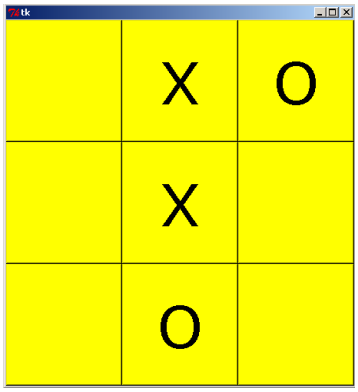
but in the short programs that we will be writing, we should be okay. Object-oriented programming provides an alternative to global variables.

15.7 Tic-tac-toe

Using Tkinter, in only about 20 lines we can make a working tic-tac-toe program:

```
from tkinter import *

def callback(r,c):
    global player
    if player == 'X':
        b[r][c].configure(text = 'X')
        player = 'O'
    else:
        b[r][c].configure(text = 'O')
        player = 'X'

root = Tk()

b = [[0,0,0],
     [0,0,0],
     [0,0,0]]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i,c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)

player = 'X'

mainloop()
```

The program works, though it does have a few problems, like letting you change a cell that already has something in it. We will fix this shortly. First, let's look at how the program does what it does. Starting at the bottom, we have a variable `player` that keeps track of whose turn it is. Above that we create the board, which consists of nine buttons stored in a two-dimensional list. We use the `lambda` trick to pass the row and column of the clicked button to the callback function. In the callback function we write an *X* or an *O* into the button that was clicked and change the value of the global variable `player`.



Correcting the problems To correct the problem about being able to change a cell that already has something in it, we need to have a way of knowing which cells have *X*'s, which have *O*'s, and which are empty. One way is to use a `Button` method to ask the button what its text is. Another way, which we will do here is to create a new two-dimensional list, which we will call `states`, that will keep track of things. Here is the code.

```
from tkinter import *

def callback(r,c):
    global player

    if player == 'X' and states[r][c] == 0:
        b[r][c].configure(text='X')
        states[r][c] = 'X'
        player = 'O'

    if player == 'O' and states[r][c] == 0:
        b[r][c].configure(text='O')
        states[r][c] = 'O'
        player = 'X'


root = Tk()

states = [[0,0,0],
          [0,0,0],
          [0,0,0]]

b = [[0,0,0],
     [0,0,0],
     [0,0,0]]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i,c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)
```

```
player = 'X'

mainloop()
```

We have not added much to the program. Most of the new action happens in the callback function. Every time someone clicks on a cell, we first check to see if it is empty (that the corresponding index in `states` is 0), and if it is, we display an *X* or *O* on the screen and record the new value in `states`. Many games have a variable like `states` that keeps track of what is on the board.

Checking for a winner We have a winner when there are three *X*'s or three *O*'s in a row, either vertically, horizontally, or diagonally. To check if there are three in a row across the top row, we can use the following if statement:

```
if states[0][0]==states[0][1]==states[0][2]!=0:
    stop_game=True
    b[0][0].configure(bg='grey')
    b[0][1].configure(bg='grey')
    b[0][2].configure(bg='grey')
```

This checks to see if each of the cells has the same nonzero entry. We are using the shortcut from Section 10.3 here in the if statement. There are more verbose if statements that would work. If we do find a winner, we highlight the winning cells and then set a global variable `stop_game` equal to **`True`**. This variable will be used in the callback function. Whenever the variable is **`True`** we should not allow any moves to take place.

Next, to check if there are three in a row across the middle row, change the first coordinate from 0 to 1 in all three references, and to check if there are three in a row across the bottom, change the 0's to 2's. Since we will have three very similar if statements that only differ in one location, a for loop can be used to keep the code short:

```
for i in range(3):
    if states[i][0]==states[i][1]==states[i][2]!=0:
        b[i][0].configure(bg='grey')
        b[i][1].configure(bg='grey')
        b[i][2].configure(bg='grey')
        stop_game = True
```

Next, checking for vertical winners is pretty much the same except we vary the second coordinate instead of the first. Finally, we have two further if statements to take care of the diagonals. The full program is at the end of this chapter. We have also added a few color options to the `configure` statements to make the game look a little nicer.

Further improvements From here it would be easy to add a restart button. The callback function for that variable should set `stop_game` back to false, it should set `states` back to all zeroes, and it should configure all the buttons back to `text=''` and `bg='yellow'`.

To add a computer player would also not be too difficult, if you don't mind it being a simple com-

puter player that moves randomly. That would take about 10 lines of code. To make an intelligent computer player is not too difficult. Such a computer player should look for two *O*'s or *X*'s in a row in order to try to win or block, as well avoid getting put into a no-win situation.

```
from tkinter import *

def callback(r,c):
    global player

    if player == 'X' and states[x][y] == 0 and stop_game==False:
        b[r][c].configure(text='X', fg='blue', bg='white')
        states[r][c] = 'X'
        player = 'O'

    if player == 'O' and states[r][c] == 0 and stop_game==False:
        b[r][c].configure(text='O', fg='orange', bg='black')
        states[r][c] = 'O'
        player = 'X'

    check_for_winner()

def check_for_winner():
    global stop_game
    for i in range(3):
        if states[i][0]==states[i][1]==states[i][2]!=0:
            b[i][0].configure(bg='grey')
            b[i][1].configure(bg='grey')
            b[i][2].configure(bg='grey')
            stop_game = True

    for i in range(3):
        if states[0][i]==states[1][i]==states[2][i]!=0:
            b[0][i].configure(bg='grey')
            b[1][i].configure(bg='grey')
            b[2][i].configure(bg='grey')
            stop_game = True

    if states[0][0]==states[1][1]==states[2][2]!=0:
        b[0][0].configure(bg='grey')
        b[1][1].configure(bg='grey')
        b[2][2].configure(bg='grey')
        stop_game = True

    if states[2][0]==states[1][1]==states[0][2]!=0:
        b[2][0].configure(bg='grey')
        b[1][1].configure(bg='grey')
        b[0][2].configure(bg='grey')
        stop_game = True

root = Tk()

b = [[0,0,0],
     [0,0,0],
```

```
     [0,0,0]]


states = [[0,0,0],
          [0,0,0],
          [0,0,0]]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i,c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)

player = 'X'
stop_game = False

mainloop()
```

Chapter 16

GUI Programming II

In this chapter we cover more basic GUI concepts.

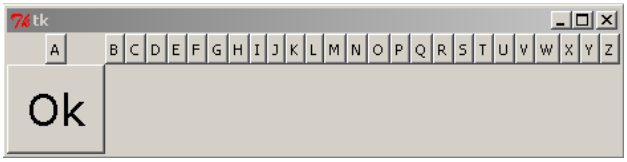
16.1 Frames

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.

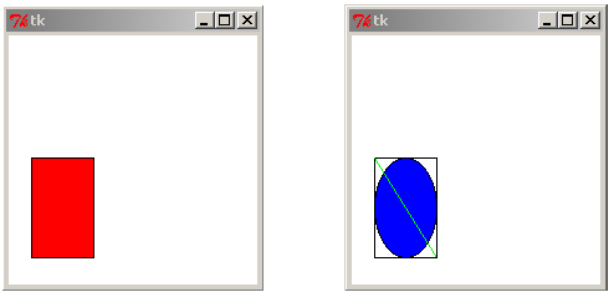
Creating canvases The following line creates a canvas with a white background that is 200 × 200 pixels in size:

```
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:

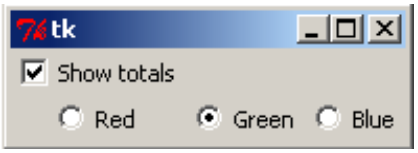
```
rect = canvas.create_rectangle(0,0,20,20)
canvas.itemconfigure(rect, fill='red')
canvas.coords(rect,40,40,60,60)
canvas.delete(rect)
```

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

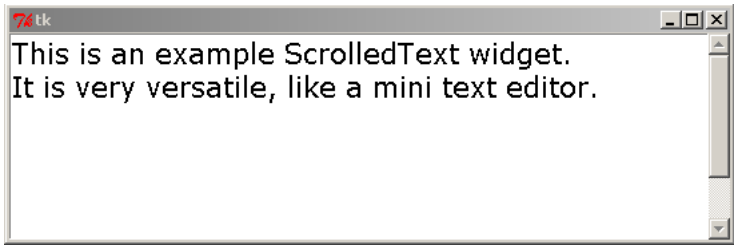
The `Text` widget is a bigger, more powerful version of the `Entry` widget. Here is an example of creating one:

```
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