Graphical User Interfaces

7. Graphical User Interfaces

Everything we have done so far has been text based. In fact, our Hangman game would not have looked out of place on a 1980s home computer. This chapter shows you how to create applications with a proper graphical user interface (GUI).

7.1. Tkinter

Tkinter is the Python interface to the Tk GUI system. Tk is not specific to Python; there are interfaces to it from many different languages, and it runs on pretty much any operating system, including Linux. Tkinter comes with Python, so there is no need to install anything. It is also the most commonly used tool for creating a GUI for Python.

7.2. Hello World

Tradition dictates that the first program you write with a new language or system should do something trivial, just to show it works! This usually means making the program display a message of "Hello World." As you'll recall, we already did this for Python back in Chapter 3, so I'll make no apologies for starting with this program:

```
#07_01_hello.py

from tkinter import *

root = Tk()

Label(root, text='Hello World').pack()

root.mainloop()
```

Figure 7-1 shows the rather unimpressive application.

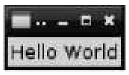


Figure 7-1. Hello World in Tkinter

You don't need to worry about how all this works. You do, however, need to know that you must assign a variable to the object `Tk`. Here, we call this variable `root`, which is a common convention. We then create an instance of the class `Label`, whose first argument is `root`. This tells Tkinter that the label belongs to it. The second argument specifies the text to display in the label. Finally, the method `pack` is called on the label. This tells the label to pack itself into the space available. The method `pack` controls the layout of the items in the window. Shortly, we will use an alternative type of layout for the components in a grid.

7.3. Temperature Converter

To get started with Tkinter, you'll gradually build up a simple application that provides a GUI for temperature conversion (see Figure 7-2). This application will use the `converter` module we created in Chapter 5 to do the calculation.

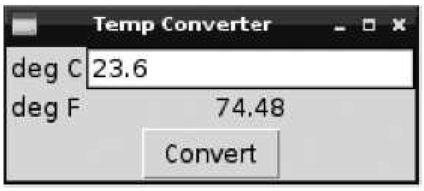


Figure 7-2. A temperature conversion application

Our Hello World application, despite being simple, is not well structured and would not lend itself well to a more complex example. It is normal when building a GUI with Tkinter to use a class to represent each application window. Therefore, our first step is to make a framework in which to slot the application, starting with a window with the title "Temp Converter" and a single label:

```
#07_02_temp_framework.py

from tkinter import *

class App:

       def __init__(self, master):

              frame = Frame(master)

              frame.pack()

              Label(frame, text='deg C').grid(row=0, column=0)

              button = Button(frame, text='Convert', command=self.convert)

              button.grid(row=1)

       def convert(self):

              print('Not implemented')

root = Tk()

root.wm_title('Temp Converter')

app = App(root)

root.mainloop()
```

We have added a class to the program called `App`. It has an `__init__` method that is used when a new instance of `App` is created in the following line:

```
app = App(root)
```

We pass in the `Tk` root object to `__init__` where the user interface is constructed.

As with the Hello World example, we are using a `Label`, but this time rather than adding the label to the root `Tk` object, we add the label to a `Frame` object that contains the label and other items that will eventually make up the window for our application. The structure of the user interface is shown in Figure 7-3. Eventually, it will have all the elements shown.

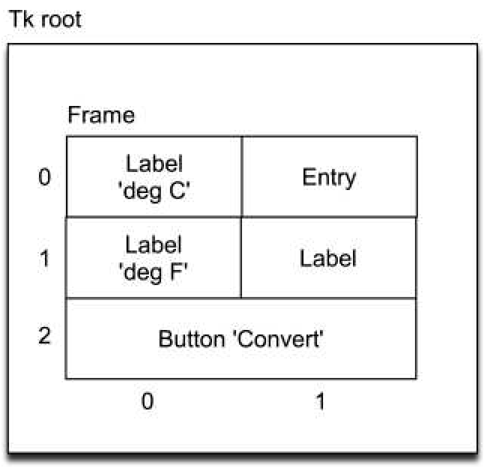


Figure 7-3. Structure of the user interface

The frame is "packed" into the root, but this time when we add the label, we use the method `grid` instead of `pack`. This allows us to specify a grid layout for the parts of our user interface. The field goes at position 0, 0 of the grid, and the button object that is created on the subsequent line is put on the second row of the grid (row 1). The button definition also specifies a "command" to be run when the button is clicked. At the moment, this is just a stub that prints the message "Not implemented."

The function `wm_title` sets the title of the window. Figure 7-4 shows what the basic user interface looks like at this point.

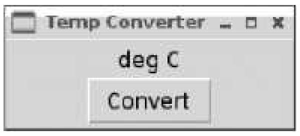


Figure 7-4. The basic user interface for the Temp Converter application

The next step is to fill in the rest of the user interface. We need an "entry" into which a value for degrees C can be entered and two more labels. We need one permanent label that just reads "deg F" and a label to the right of it where the converted temperature will be displayed.

Tkinter has a special way of linking fields on the user interface with values. Therefore, when we need to get or set the value entered or displayed on a label or entry, we create an instance of a special variable object. This comes in various flavors, and the most common is StringVar. However, because we are entering and displaying numbers, we will use DoubleVar. *Double* means a double-precision floating-point number. This is just like a float, but more precise.

After we add in the rest of the user interface controls and the variables to interact with them, the program will look like this:

```
#07_03_temp_ui.py

from tkinter import *

class App:

    def __init__(self, master):

        frame = Frame(master)

        frame.pack()

        Label(frame, text='deg C').grid(row=0, column=0)

        self.c_var = DoubleVar()

        Entry(frame, textvariable=self.c_var).grid(row=0, column=1)

        Label(frame, text='deg F').grid(row=1, column=0)

        self.result_var = DoubleVar()

        Label(frame, textvariable=self.result_var).grid(row=1, column=1)

        button = Button(frame, text='Convert', command=self.convert)

        button.grid(row=2, columnspan=2)

    def convert(self):

        print('Not implemented')

root = Tk()

root.wm_title('Temp Converter')

app = App(root)

root.mainloop()
```

The first DoubleVar (`c_var`) is assigned to the entry by specifying a `textvariable` property for it. This means that the entry will display what is in that DoubleVar, and if the value in the DoubleVar is changed, the field display will automatically update to show the new value. Also, when the user types something in the entry field, the value in the DoubleVar will change. Note that a new label of "deg F" has also been added.

The second DoubleVar is linked to another label that will eventually display the result of the calculation. We have added another attribute to the `grid` command that lays out the button. Because we specify `columnspan=2`, the button will stretch across both columns.

If you run the program, it will display the final user interface, but when you click the Convert button, the message "Not Implemented" will be written to the Python console.

The last step is to replace the stubbed-out "`convert`" method with a real method that uses the `converters` module from Chapter 5. To do this, we need to import the module. In order to reduce how much we need to type, we will import everything, as follows:

```
from converters import *
```

For the sake of efficiency, it is better if we create a single "converter" during `__init__` and just use the same one every time the button is clicked. Therefore, we create a variable called `self.t_conv` to reference the convertor. The `convert` method then just becomes this:

```
def convert(self):

              c = self.c_var.get()

              self.result_var.set(self.t_conv.convert(c))
```

Here is the full listing of the program:

```
#07_04_temp_final.py

from tkinter import *

from converters import *

class App:

     def __init__(self, master):

        self.t_conv = ScaleAndOffsetConverter('C', 'F', 1.8, 32)

        frame = Frame(master)

        frame.pack()

        Label(frame, text='deg C').grid(row=0, column=0)

        self.c_var = DoubleVar()

        Entry(frame, textvariable=self.c_var).grid(row=0, column=1)

        Label(frame, text='deg F').grid(row=1, column=0)

        self.result_var = DoubleVar()

        Label(frame, textvariable=self.result_var).grid(row=1, column=1)

        button = Button(frame, text='Convert', command=self.convert)

        button.grid(row=2, columnspan=2)

     def convert(self):

        c = self.c_var.get()

        self.result_var.set(self.t_conv.convert(c))

root = Tk()

root.wm_title('Temp Converter')
```

```
app = App(root)

root.mainloop()
```

7.4. Other GUI Widgets

In the temperature converter, we just used text fields (class `Entry`) and labels (class `Label`). As you would expect, you can build lots of other user interface controls into your application. Figure 7-5 shows the main screen of a "kitchen sink" application that illustrates most of the controls you can use in Tkinter. This program is available as 07_05_kitchen_sink.py.

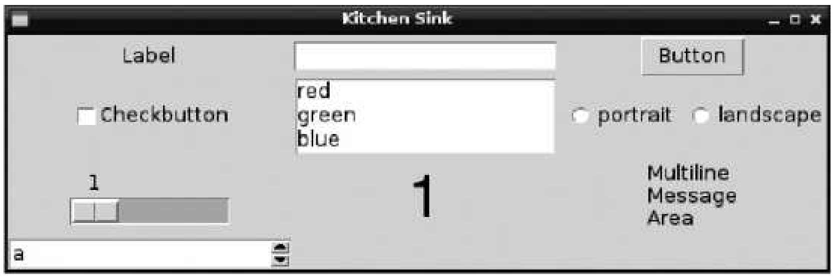


Figure 7-5. A "kitchen sink" application

7.4.1. Checkbutton

The Checkbox widget (first column, second row of Figure 7-5) is created like this:

```
Checkbutton(frame, text='Checkbutton')
```

This line of code just creates a Checkbutton with a label next to it. If we have gone to the effort of placing a check box on the window, we'll also want a way of finding out whether or not it is checked.

The way to do this is to use a special "variable" like we did in the temperature converter example. In the following example, we use a StringVar, but if the values of `onvalue` and `offvalue` were numbers, we could use an IntVar instead.

```
check_var = StringVar()

check = Checkbutton(frame, text='Checkbutton',

            variable=check_var, onvalue='Y', offvalue='N')

check.grid(row=1, column=0)
```

7.4.2. Listbox

To display a list of items from which one or multiple items can be selected, a Listbox is used (refer to the center of Figure 7-5). Here's an example:

```
listbox = Listbox(frame, height=3, selectmode=BROWSE)

for item in ['red', 'green', 'blue', 'yellow', 'pink']:

              listbox.insert(END, item)

listbox.grid(row=1, column=1)
```

In this case, it just displays a list of colors. Each string has to be added to the list individually. The word `END` indicates that the item should go at the end of the list.

You can control the way selections are made on the Listbox using the `selectmode` property, which can be set to one of the following:

1. **`SINGLE`** Only one selection at a time.
2. **`BROWSE`** Similar to `SINGLE`, but allows selection using the mouse. This appears to be indistinguishable from `SINGLE` in Tkinter on the Pi.

3. **`MULTIPLE`** SHIFT-click to select more than one row.

4. **`EXTENDED`** Like `MULTIPLE`, but also allows the CTRL-SHIFT-click selection of ranges.

Unlike with other widgets that use StringVar or some other type of special variable to get values in and out, to find out which items of the Listbox are selected, you have to ask it using the method `curselection`. This returns a collection of selection indexes. Thus, if the first, second, and fourth items in the list are selected, you will get a list like this:

```
[0, 1, 3]
```

When `selectmode` is `SINGLE`, you still get a list back, but with just one value in it.

7.4.3. Spinbox

Spinboxes provide an alternative way of making a single selection from a list:

```
Spinbox(frame, values=('a','b','c')).grid(row=3)
```

The `get` method returns the currently displayed item in the Spinbox, not its selection index.

7.4.4. Layouts

Laying out the different parts of your application so that everything looks good, even when you resize the window, is one of the most tricky parts of building a GUI.

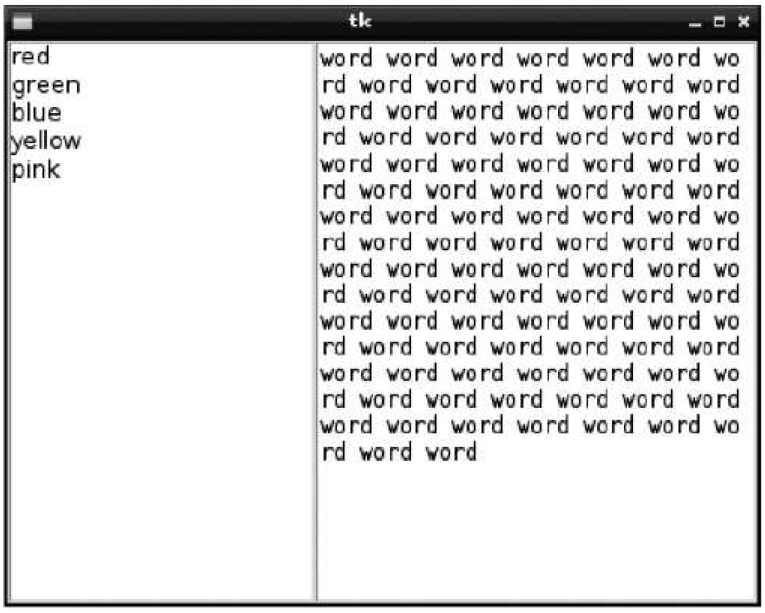
You will often find yourself putting one kind of layout inside another. For example, the overall shape of the "kitchen sink" application is a 3×3 grid, but within that grid is another frame for the two radio buttons:

```
radio_frame = Frame(frame)

radio_selection = StringVar()

b1 = Radiobutton(radio_frame, text='portrait',

       variable=radio_selection, value='P')

b1.pack(side=LEFT)

b2 = Radiobutton(radio_frame, text='landscape',

       variable=radio_selection, value='L')

b2.pack(side=LEFT)

radio_frame.grid(row=1, column=2)
```

This approach is quite common, and it is a good idea to sketch out the layout of your controllers on paper before you start writing the code.

One particular problem you will encounter when creating a GUI is controlling what happens when the window is resized. You will normally want to keep some widgets in the same place and at the same size, while allowing other widgets to expand.

As an example of this, we can build a simple window like the one shown in Figure 7-6, which has a Listbox (on the left) that stays the same size and an expandable message area (on the right) that expands as the window is resized.



```
red
green
blue
yellow
pink
```

```
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word word word word word
word word word word word word wo
rd word word
```

Figure 7-6. An example of resizing a window

The code for this is shown here:

```
#07_06_resizing.py

from tkinter import *

class App:

       def __init__(self, master):

              frame = Frame(master)

              frame.pack(fill=BOTH, expand=1)

              #Listbox

              listbox = Listbox(frame)

              for item in ['red', 'green', 'blue', 'yellow', 'pink']:

                     listbox.insert(END, item)

              listbox.grid(row=0, column=0, sticky=W+E+N+S)

              #Message

              text = Text(frame, relief=SUNKEN)

              text.grid(row=0, column=1, sticky=W+E+N+S)

              text.insert(END, 'word ' * 100)

              frame.columnconfigure(1, weight=1)

              frame.rowconfigure(0, weight=1)

root = Tk()

app = App(root)

root.geometry("400x300+0+0")

root.mainloop()
```

The key to understanding such layouts is the use of the `sticky` attributes of the components to decide which walls of their grid cell they should stick to. To control which of the columns and rows expand when the window is resized, you use the `columnconfigure` and `rowconfigure` commands. Figure 7-7 shows the arrangement of GUI components that make up this window. The lines indicate where the edge of a user interface item is required to "stick" to its containing wall.

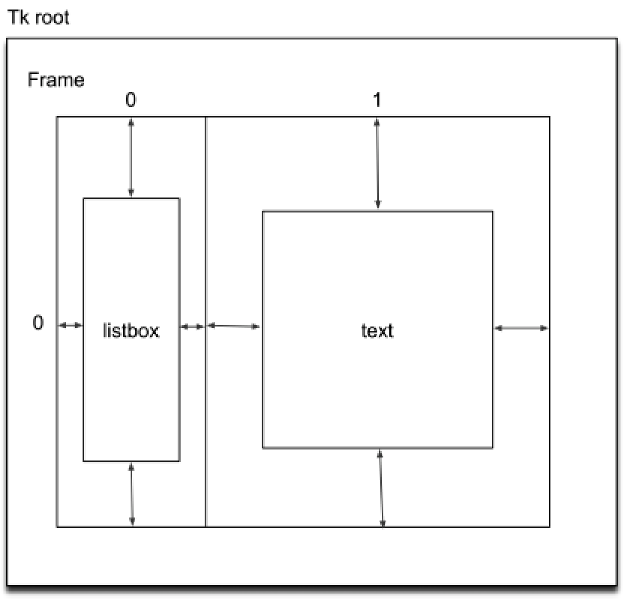


Figure 7-7. Layout for the resizing window example

Let's go through the code for this example so that things start to make sense. First, the line

```
frame.pack(fill=BOTH, expand=1)
```

ensures that the frame will fill the enclosing root window so that if the root window changes in size, so will the frame.

Having created the Listbox, we add it to the frame's grid layout using the following line:

```
listbox.grid(row=0, column=0, sticky=W+E+N+S)
```

This specifies that the Listbox should go in position row 0, column 0, but then the `sticky` attribute says that the west, east, north, and south sides of the Listbox should stay connected to the enclosing grid. The constants `W`, `E`, `N`, and `S` are numeric constants that can be added together in any order. The Text widget is added to the frame's grid in just the same way, and its content is initialized to the word *word* repeated 100 times.

The final part to the puzzle is getting the resizing behavior we want for a text area that expands to the right and a list area that doesn't. To do this, we use the `columnconfigure` and `rowconfigure` methods:

```
frame.columnconfigure(1, weight=1)

frame.rowconfigure(0, weight=1)
```

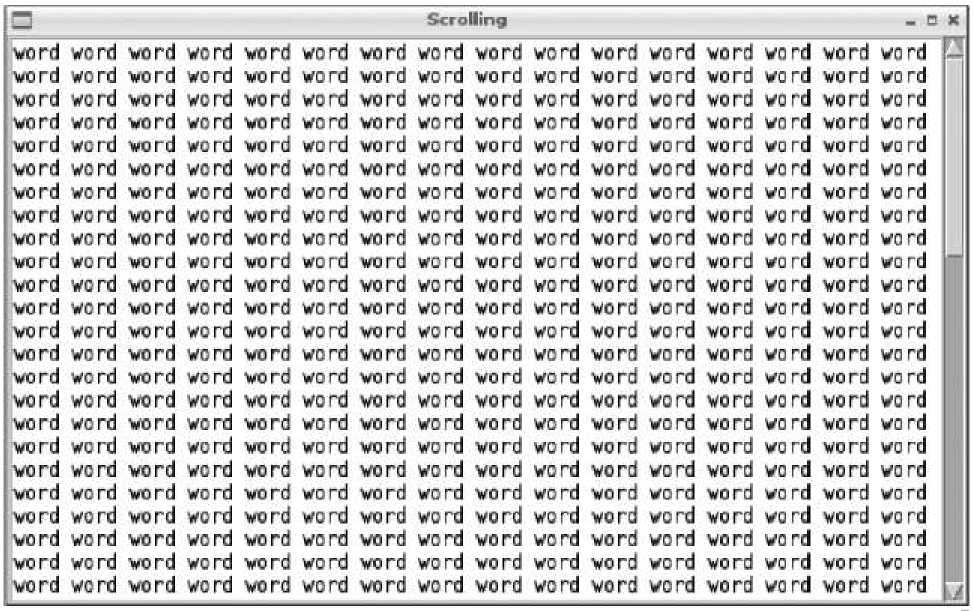
By default, rows and columns do not expand at all when their enclosing user interface element expands. We do not want column 0 to expand, so we can leave that alone. However, we do want column 1 to expand to the right, and we want row 0 (the only row) to be able to expand downward. We do this by giving them a "weight" using the `columnconfigure` and `rowconfigure` methods. If, for example, we had multiple columns that we want to expand evenly, we would give them the same weight (typically `1`). If, however, we want one of the columns to expand at twice the rate of the other, we would give it twice the weight. In this case, we only have one column and one row that we need expanding, so they can both be given a weight of `1`.

7.4.5. Scrollbar

If you shrink down the window for the program 07_06_resizing.py, you will notice that there's no scrollbar to access text that's hidden. You can still get to the text, but clearly a scrollbar would help.

Scrollbars are widgets in their own right, and the trick for making them work with something like a Text, Message, or Listbox widget is to lay them out next to each other and then link them together.

Figure 7-8 shows a Text widget with a scrollbar.



```
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
word word word word word word word word word word word word word word word word
```

Figure 7-8. Scrolling a Text widget

The code for this is as follows:

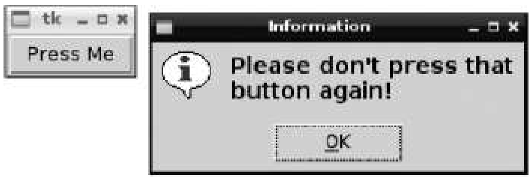
```
#07_07_scrolling.py

from tkinter import *

class App:

       def __init__(self, master):

              scrollbar = Scrollbar(master)

              scrollbar.pack(side=RIGHT, fill=Y)

              text = Text(master, yscrollcommand=scrollbar.set)

              text.pack(side=LEFT, fill=BOTH)

              text.insert(END, 'word ' * 1000)

              scrollbar.config(command=text.yview)

root = Tk()

root.wm_title('Scrolling')

app = App(root)

root.mainloop()
```

In this example, we use the `pack` layout, positioning the scrollbar on the right and the text area on the left. The `fill` attribute specifies that the Text widget is allowed to use all free space on *both* the X and Y dimensions.

To link the scrollbar to the Text widget, we set the `yscrollcommand` property of the Text widget to the `set` method of the scrollbar. Similarly, the `command` attribute of the scrollbar is set to `text.yview`.

7.5. Dialogs

It is sometimes useful to pop up a little window with a message and make the user click OK before they can do anything else (see Figure 7-9). These windows are called *modal dialogs*, and Tkinter has a whole range of them in the package `tkinter.messagebox`.



Please don't press that button again!

Figure 7-9. An alert dialog

The following example shows how to display such an alert. As well as `showinfo`, `tkinter.messagebox` also has the functions `showwarning` and `showerror` that work just the same, but display a different symbol in the window.

```
#07_08_gen_dialogs.py

from tkinter import *

import tkinter.messagebox as mb

class App:

     def __init__(self, master):

          b=Button(master, text='Press Me', command=self.info).pack()

     def info(self):

          mb.showinfo('Information', "Please don't press that button again!")

root = Tk()

app = App(root)

root.mainloop()
```

Other kinds of dialogs can be found in the packages `tkinter.colorchooser` and `tkinter.filedialog`.

7.5.1. Color Chooser

The Color Chooser returns a color as separate RGB components as well as a standard hex color string (see Figure 7-10).

```
#07_09_color_chooser.py

from tkinter import *

import tkinter.colorchooser as cc

class App:

     def __init__(self, master):

         b=Button(master, text='Color..', command=self.ask_color).pack()

     def ask_color(self):

          (rgb, hx) = cc.askcolor()

          print("rgb=" + str(rgb) + " hx=" + hx)

root = Tk()

app = App(root)

root.mainloop()
```

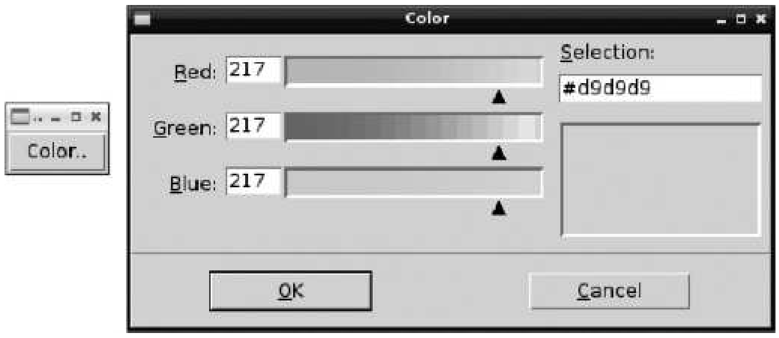


Figure 7-10. The Color Chooser

This code returns something like this:

```
rgb=(255.99609375, 92.359375, 116.453125) hx=#ff5c74
```

7.5.2. File Chooser

File Choosers can be found in the package `tkinter.filedialog`. These follow exactly the same pattern as the other dialogs we have looked at.

7.6. Menus

You can give your applications menus. As an example, we can create a very simple application with an entry field and a couple of menu options (see Figure 7-11).

```
#07_10_menus.py

from tkinter import *

class App:

       def __init__(self, master):

              self.entry_text = StringVar()

              Entry(master, textvariable=self.entry_text).pack()

              menubar = Menu(root)

              filemenu = Menu(menubar, tearoff=0)

              filemenu.add_command(label='Quit', command=exit)

              menubar.add_cascade(label='File', menu=filemenu)

              editmenu = Menu(menubar, tearoff=0)

              editmenu.add_command(label='Fill', command=self.fill)

              menubar.add_cascade(label='Edit', menu=editmenu)

              master.config(menu=menubar)

       def fill(self):

              self.entry_text.set('abc')

root = Tk()

app = App(root)

root.mainloop()
```

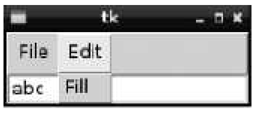


Figure 7-11. Menus

The first step is to create a root `Menu`. This is the single object that will contain all the menus (File and Edit, in this case, along with all the menu options).

```
menubar = Menu(root)
```

To create the File menu, with its single option, Quit, we first create another instance of `Menu` and then add a command for Quit and finally add the File menu to the root `Menu`:

```
filemenu = Menu(menubar, tearoff=0)

filemenu.add_command(label='Quit', command=exit)

menubar.add_cascade(label='File', menu=filemenu)
```

The Edit menu is created in just the same way. To make the menus appear on the window, we have to use the following command:

```
master.config(menu=menubar)
```

7.7. The Canvas

In the next chapter, you'll get a brief introduction to game programming using PyGame. This allows all sorts of nice graphical effects to be achieved. However, if you just need to create simple graphics, such as drawing shapes or plotting line graphs on the screen, you can use Tkinter's Canvas interface instead (see Figure 7-12).

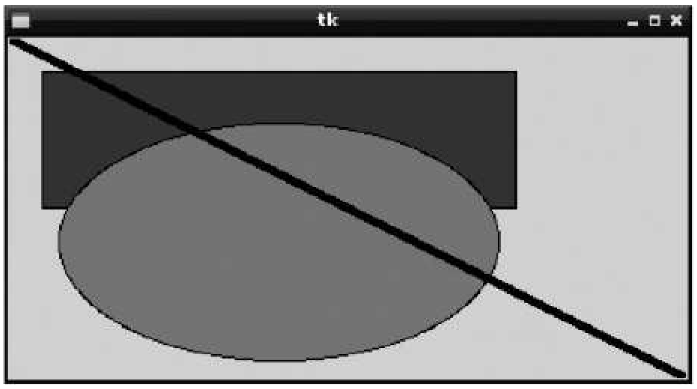


Figure 7-12. The Canvas widget

The Canvas is just like any other widget you can add to a window. The following example shows how to draw rectangles, ovals, and lines:

```
  #07_11_canvas.py

from tkinter import *

class App:

       def __init__(self, master):

              canvas = Canvas(master, width=400, height=200)

              canvas.pack()

              canvas.create_rectangle(20, 20, 300, 100, fill='blue')

              canvas.create_oval(30, 50, 290, 190, fill='#ff2277')

              canvas.create_line(0, 0, 400, 200, fill='black', width=5)
```

```
root = Tk()

app = App(root)

root.mainloop()
```

You can draw arcs, images, polygons, and text in a similar way. Refer to an online Tkinter reference such as `http://infohost.nmt.edu/tcc/help/pubs/tkinter/` for more information.

NOTE *The origin of the coordinates is the top-left corner of the window, and the coordinates are in pixels.*

7.8. Summary

In a book this size, it is sometimes only possible to introduce a topic and get you started on the right path. Once you've followed the examples in this chapter, run them, altered them, and analyzed what's going on, you will soon find yourself hungry for more information. You will get past the need for hand-holding and have specific ideas of what you want to write. No book is going to tell you exactly how to build the project you have in your head. This is where the Internet really comes into its own.

Good online references to take what you've learned further can be found here:

1. `www.pythonware.com/library/tkinter/introduction/`
2. `http://infohost.nmt.edu/tcc/help/pubs/tkinter/`

Citation

EXPORT

Dr. Simon Monk: Programming the Raspberry Pi: Getting Started with Python. Graphical User Interfaces, Chapter (McGraw-Hill Professional, 2013), AccessEngineering