CHAPTER 16

Creating Graphical User Interfaces

Most of the programs in previous chapters are not interactive. Once launched, they run to completion without giving us a chance to steer them or provide new input. The few that do communicate with us do so through the kind of text-only *command-line user interface*, or CLUI, that would have already been considered old-fashioned in the early 1980s.

As you already know, most modern programs interact with users via a *graphical user interface*, or GUI, which is made up of windows, menus, buttons, and so on. In this chapter, we will show you how to build simple GUIs using a Python module called `tkinter`. Along the way, we will introduce a different way of structuring programs called *event-driven programming*. A traditionally structured program usually has control over what happens when, but an event-driven program must be able to respond to input at unpredictable moments.

`tkinter` is one of several toolkits you can use to build GUIs in Python. It is the only one that comes with a standard Python installation.

16.1 Using Module Tkinter

Every `tkinter` program consists of these things:

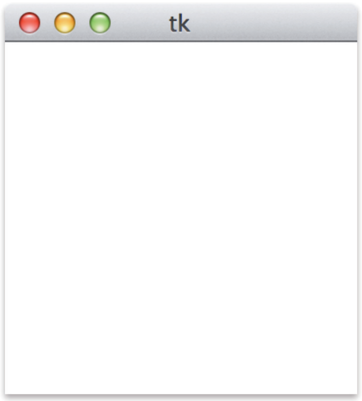
- Windows, buttons, scrollbars, text areas, and other *widgets*—anything that you can see on the computer screen (Generally, the term *widget* means any useful object; in programming, it is short for “window gadget.”)
- Modules, functions, and classes that manage the data that is being shown in the GUI—you are familiar with these; they are the tools you’ve seen so far in this book.
- An event manager that *listens* for events such as mouse clicks and keystrokes and reacts to these events by calling event handler functions

Here is a small but complete tkinter program:

```
import tkinter
window = tkinter.Tk()
window.mainloop()
```

Tk is a class that represents the *root window* of a tkinter GUI. This root window's mainloop method handles all the events for the GUI, so it's important to create only one instance of Tk.

Here is the resulting GUI:



The root window is initially empty; you'll see in the next section how to add widgets to it. If the window on the screen is closed, the window object is destroyed (though we can create a new root window by calling Tk() again). All of the applications we will create have only one root window, but additional windows can be created using the TopLevel widget.

The call on method mainloop doesn't exit until the window is destroyed (which happens when you click the appropriate widget in the title bar of the window), so any code following that call won't be executed until later:

```
import tkinter
window = tkinter.Tk()
window.mainloop()
print('Anybody home?')
```

When you try this code, you'll see that the call on function print doesn't get executed until after the window is destroyed. That means that if you want to make changes to the GUI after you have called mainloop, you need to do it in an event-handling function.

The following table gives a list of some of the available tkinter widgets:

Widget	Description
Button	A clickable button
Canvas	An area used for drawing or displaying images
Checkbutton	A clickable box that can be selected or unselected
Entry	A single-line text field that the user can type in
Frame	A container for widgets
Label	A single-line display for text
Listbox	A drop-down list that the user can select from
Menu	A drop-down menu
Message	A multiline display for text
Menubutton	An item in a drop-down menu
Text	A multiline text field that the user can type in
TopLevel	An additional window

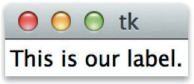
Table 26—tkinter Widgets

16.2 Building a Basic GUI

Labels are widgets that are used to display short pieces of text. Here we create a Label that belongs to the root window—its *parent widget*—and we specify the text to be displayed by assigning it to the Label's text parameter.

```
import tkinter

window = tkinter.Tk()
label = tkinter.Label(window, text='This is our label.')
label.pack()

window.mainloop()
```

Here is the resulting GUI:



This is our label.

Method call label.pack() is crucial. Each widget has a method called pack that places it in its parent widget and then tells the parent to resize itself as necessary. If we forget to call this method, the child widget (in this case, Label) won't be displayed or will be displayed improperly.

Labels display text. Often, applications will want to update a label's text as the program runs to show things like the name of a file or the time of day. One way to do this is simply to assign a new value to the widget's text using method config:

```
import tkinter

window = tkinter.Tk()
label = tkinter.Label(window, text='First label.')
label.pack()
label.config(text='Second label.')
```

Run the previous code one line at a time from the Python shell to see how the label changes. (This code will not display the window at all if you run it as a program because we haven't called method mainloop.)

Using Mutable Variables with Widgets

Suppose you want to display a string, such as the current time or a score in a game, in several places in a GUI—the application's status bar, some dialog boxes, and so on. Calling method config on each widget every time there is new information isn't hard, but as the application grows, so too do the odds that we'll forget to update at least one of the widgets that's displaying the string. What we really want is a string that "knows" which widgets care about its value and can alert them itself when that value changes.

Python's strings, integers, floating-point numbers, and Booleans are immutable, so module tkinter provides one class for each of the immutable types: StringVar for str, IntVar for int, BooleanVar for bool, and DoubleVar for float. (The use of the word *double* is historical; it is short for "double-precision floating-point number.") These mutable types can be used instead of the immutable ones; here we show how to use a StringVar instead of a str:

```
import tkinter

window = tkinter.Tk()
data = tkinter.StringVar()
data.set('Data to display')
label = tkinter.Label(window, textvariable=data)
label.pack()

window.mainloop()
```

Notice that this time we assign to the textvariable parameter of the label rather than the text parameter.

The values in tkinter containers are set and retrieved using the methods set and get. Whenever a set method is called, it tells the label, and any other widgets it has been assigned to, that it's time to update the GUI.

There is one small trap here for newcomers: because of the way module tkinter is structured, you cannot create a StringVar or any other mutable variable until you have created the root Tk window.

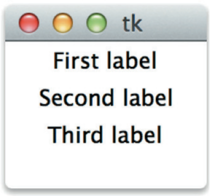
Grouping Widgets with the Frame Type

A tkinter Frame is a container, much like the root window is a container. Frames are not directly visible on the screen; instead, they are used to organize other widgets. The following code creates a frame, puts it in the root window, and then adds three Labels to the frame:

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
first = tkinter.Label(frame, text='First label')
first.pack()
second = tkinter.Label(frame, text='Second label')
second.pack()
third = tkinter.Label(frame, text='Third label')
third.pack()

window.mainloop()
```

Note that we call pack on every widget; if we omit one of these calls, that widget will not be displayed.

Here is the resulting GUI:



First label
Second label
Third label

In this particular case, putting the three Labels in a frame looks the same as when we put the Labels directly into the root window. However, with a more complicated GUI, we can use multiple frames to format the window's content and layout.

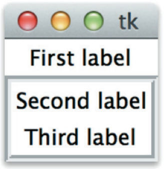
Here's an example with the same three Labels but with two frames instead of one. The second frame has a visual border around it:

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
frame2 = tkinter.Frame(window, borderwidth=4, relief=tkinter.GROOVE)
frame2.pack()
first = tkinter.Label(frame, text='First label')
first.pack()
```

```
second = tkinter.Label(frame2, text='Second label')
second.pack()
third = tkinter.Label(frame2, text='Third label')
third.pack()

window.mainloop()
```

We specify the border width using the borderwidth keyword argument (0 is the default) and the border style using relief (FLAT is the default). The other border styles are SUNKEN, RAISED, GROOVE, and RIDGE.

Here is the resulting GUI:



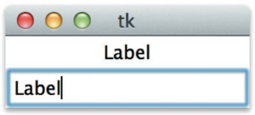
Getting Information from the User with the Entry Type

Two widgets let users enter text. The simplest one is Entry, which allows for a single line of text. If we associate a StringVar with the Entry, then whenever a user types anything into that Entry, the StringVar's value will automatically be updated to the contents of the Entry.

Here's an example that associates a single StringVar with both a Label and an Entry. When the user enters text in the Entry, the StringVar's contents will change. This will cause the Label to be updated, and so the Label will display whatever is currently in the Entry.

```
import tkinter
window = tkinter.Tk()

frame = tkinter.Frame(window)
frame.pack()
var = tkinter.StringVar()
label = tkinter.Label(frame, textvariable=var)
label.pack()
entry = tkinter.Entry(frame, textvariable=var)
entry.pack()
window.mainloop()
```

Here is the resulting GUI:



16.3 Models, Views, and Controllers, Oh My!

Using a StringVar to connect a text-entry box and a label is the first step toward separating *models* (How do we represent the data?), *views* (How do we display the data?), and *controllers* (How do we modify the data?), which is the key to building larger GUIs (as well as many other kinds of applications). This MVC design helps separate the parts of an application, which will make the application easier to understand and modify. The main goal of this design is to keep the representation of the data separate from the parts of the program that the user interacts with; that way, it is easier to make changes to the GUI code without affecting the code that manipulates the data.

As its name suggests, a view is something that displays information to the user, like Label. Many views, like Entry, also accept input, which they display immediately. The key is that they don't do anything else: they don't calculate average temperatures, move robot arms, or do any other calculations.

Models, on the other hand, store data, like a piece of text or the current inclination of a telescope. They also don't do calculations; their job is simply to keep track of the application's current state (and, in some cases, to save that state to a file or database and reload it later).

Controllers are the pieces that convert user input into calls on functions in the model that manipulate the data. The controller is what decides whether two gene sequences match well enough to be colored green or whether someone is allowed to overwrite an old results file. Controllers may update an application's models, which in turn can trigger changes to its views.

The following code shows what all of this looks like in practice. Here the model is kept track of by variable counter, which refers to an IntVar so that the view will update itself automatically. The controller is function click, which updates the model whenever a button is clicked. Four objects make up the view: the root window, a Frame, a Label that shows the current value of counter, and a button that the user can click to increment the counter's value:

```
import tkinter

# The controller.
def click():
    counter.set(counter.get() + 1)

if __name__ == '__main__':
    window = tkinter.Tk()
    # The model.
    counter = tkinter.IntVar()
    counter.set(0)
    # The views.
    frame = tkinter.Frame(window)
    frame.pack()

    button = tkinter.Button(frame, text='Click', command=click)
    button.pack()

    label = tkinter.Label(frame, textvariable=counter)
    label.pack()

    # Start the machinery!
    window.mainloop()
```

The first two arguments used to construct the Button should be familiar by now. The third, command=click, tells it to call function click each time the user presses the button. This makes use of the fact that in Python a function is just another kind of object and can be passed as an argument like anything else.

Function click in the previous code does not have any parameters but uses variable counter, which is defined outside the function. Variables like this are called *global variables*, and their use should be avoided, since they make programs hard to understand. It would be better to pass any variables the function needs into it as parameters. We can't do this using the tools we have seen so far, because the functions that our buttons can call must not have any parameters. We will show you one way to avoid using global variables in the next section.

Using Lambda

The simple counter GUI shown earlier does what it's supposed to, but there is room for improvement. For example, suppose we want to be able to lower the counter's value as well as raise it.

Using only the tools we have seen so far, we could add another button and another controller function like this:

```
import tkinter

window = tkinter.Tk()

# The model.
counter = tkinter.IntVar()
counter.set(0)

# Two controllers.
def click_up():
    counter.set(counter.get() + 1)
```

```
def click_down():
    counter.set(counter.get() - 1)

# The views.
frame = tkinter.Frame(window)
frame.pack()
button = tkinter.Button(frame, text='Up', command=click_up)
button.pack()
button = tkinter.Button(frame, text='Down', command=click_down)
button.pack()
label = tkinter.Label(frame, textvariable=counter)
label.pack()

window.mainloop()
```

This seems a little clumsy, though. Functions `click_up` and `click_down` are doing almost the same thing; surely we ought to be able to combine them into one. While we're at it, we'll pass `counter` into the function explicitly rather than using it as a global variable:

```
# The model.
counter = tkinter.IntVar()
counter.set(0)

# One controller with parameters.
def click(variable, value):
    variable.set(variable.get() + value)
```

The problem with this is figuring out what to pass into the buttons, since we can't provide any arguments for the functions assigned to the buttons' `command` keyword arguments when creating those buttons. `tkinter` cannot read our minds—it can't magically know how many arguments our functions require or what values to pass in for them. For that reason, it requires that the controller functions triggered by buttons and other widgets take zero arguments so they can all be called the same way. It is our job to figure out how to take the two-argument function we want to use and turn it into one that needs no arguments at all.

We *could* do this by writing a couple of wrapper functions:

```
def click_up():
    click(counter, 1)

def click_down():
    click(counter, -1)
```

But this gets us back to two nearly identical functions that rely on global variables. A better way is to use a *lambda function*, which allows us to create a one-line function anywhere we want without giving it a name. Here's a very simple example:

```
>>> lambda: 3
<function <lambda> at 0x00A89B30>
>>> (lambda: 3)()
3
```

The expression lambda: 3 on the first line creates a nameless function that always returns the number 3. The second expression creates this function and immediately calls it, which has the same effect as this:

```
>>> def f():
...     return 3
...
>>> f()
3
```

However, the lambda form does *not* create a new variable or change an existing one. Finally, lambda functions can take arguments, just like other functions:

```
>>> (lambda x: 2 * x)(3)
6
```

Why *Lambda*?

The name *lambda function* comes from lambda calculus, a mathematical system for investigating function definition and application that was developed in the 1930s by Alonzo Church and Stephen Kleene.

So how does this help us with GUIs? The answer is that it lets us write one controller function to handle different buttons in a general way and then wrap up calls to that function when and as needed. Here's the two-button GUI once again using lambda functions:

```
import tkinter

window = tkinter.Tk()

# The model.
counter = tkinter.IntVar()
counter.set(0)

# General controller.
def click(var, value):
    var.set(var.get() + value)

# The views.
frame = tkinter.Frame(window)
frame.pack()
```

```
button = tkinter.Button(frame, text='Up', command=lambda: click(counter, 1))
button.pack()

button = tkinter.Button(frame, text='Down', command=lambda: click(counter, -1))
button.pack()

label = tkinter.Label(frame, textvariable=counter)
label.pack()

window.mainloop()
```

This code creates a zero-argument lambda function to pass into each button just where it's needed. Those lambda functions then pass the right values into click. This is cleaner than the preceding code because the function definitions are enclosed in the call that uses them—there is no need to clutter the GUI with little functions that are used only in one place.

Note, however, that it is a very bad idea to repeat the same function several times in different places—if you do that, the odds are very high that you will one day want to change them all but will miss one or two. If you find yourself wanting to do this, reorganize the code so that the function is defined only once.

16.4 Customizing the Visual Style

Every windowing system has its own *look and feel*—square or rounded corners, particular colors, and so on. In this section, we'll see how to change the appearance of GUI widgets to make applications look more distinctive.

A note of caution before we begin: the default styles of some windowing systems have been chosen by experts trained in graphic design and human-computer interaction. The odds are that any radical changes on your part will make things worse, not better. In particular, be careful about color (several percent of the male population has some degree of color blindness) and font size (many people, particularly the elderly, cannot read small text).

Changing Fonts

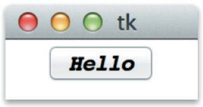
Let's start by changing the size, weight, slant, and family of the font used to display text. To specify the size, we provide the height as an integer in points. We can set the weight to either bold or normal and the slant to either italic (slanted) or roman (not slanted).

The font families we can use depend on what system the program is running on. Common families include Times, Courier, and Verdana, but dozens of others are usually available. One note of caution though: if you choose an unusual font, people running your program on other computers might not have it, so your GUI might appear different than you'd like for them. Every operating system has a default font that will be used if the requested font isn't installed.

The following sets the font of a button to be 14 point, bold, italic, and Courier.

```
import tkinter

window = tkinter.Tk()
button = tkinter.Button(window, text='Hello',
                        font=('Courier', 14, 'bold italic'))
button.pack()
window.mainloop()
```

Here is the resulting GUI:



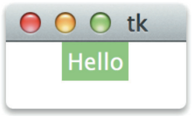
Using this technique, you can set the font of any widget that displays text.

Changing Colors

Almost all background and foreground colors can be set using the bg and fg keyword arguments, respectively. As the following code shows, we can set either of these to a standard color by specifying the color's name, such as white, black, red, green, blue, cyan, yellow, or magenta:

```
import tkinter

window = tkinter.Tk()
button = tkinter.Label(window, text='Hello', bg='green', fg='white')
button.pack()
window.mainloop()
```

Here is the resulting GUI:



As you can see, white text on a bright green background is *not* particularly readable.

We can choose more colors by specifying them using the *RGB color model*. RGB is an abbreviation for "red, green, blue"; it turns out that every color can be created using different amounts of these three colors. The amount of each color is usually specified by a number between 0 and 255 (inclusive).

These numbers are conventionally written in hexadecimal (base 16) notation; the best way to understand them is to play with them. Base 10 uses the digits 0 through 9; base 16 uses those ten digits plus another six: A, B, C, D, E, and F. In base 16, the number 255 is written FF.

The following color picker does this by updating a piece of text to show the color specified by the red, green, and blue values entered in the text boxes; choose any two base-16 digits for the RGB values and click the Update button:

```
import tkinter
def change(widget, colors):
    """ Update the foreground color of a widget to show the RGB color value
    stored in a dictionary with keys 'red', 'green', and 'blue'.  Does
    *not* check the color value.
    """

    new_val = '#'
    for name in ('red', 'green', 'blue'):
        new_val += colors[name].get()
    widget['bg'] = new_val

# Create the application.
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()

# Set up text entry widgets for red, green, and blue, storing the
# associated variables in a dictionary for later use.
colors = {}
for (name, col) in (('red', '#FF0000'),
                    ('green', '#00FF00'),
                    ('blue', '#0000FF')):
    colors[name] = tkinter.StringVar()
    colors[name].set('00')
    entry = tkinter.Entry(frame, textvariable=colors[name], bg=col,
                          fg='white')
    entry.pack()

# Display the current color.
current = tkinter.Label(frame, text='     ', bg='#FFFFFF')
current.pack()

# Give the user a way to trigger a color update.
update = tkinter.Button(frame, text='Update',
                        command=lambda: change(current, colors))
update.pack()
tkinter.mainloop()
```

This is the most complicated GUI we have seen so far, but it can be understood by breaking it down into a model, some views, and a controller. The model is three StringVars that store the hexadecimal strings representing the current red, green, and blue components of the color to display. These three variables are kept in a dictionary indexed by name for easy access. The controller is function change, which concatenates the strings to create an RGB color and applies that color to the background of a widget. The views are the text-entry boxes for the color components, the label that displays the current color, and the button that tells the GUI to update itself.

This program works, but neither the GUI nor the code is very attractive. It's annoying to have to click the update button, and if a user ever types anything that isn't a two-digit hexadecimal value into one of the text boxes, it results in an error. The exercises will ask you to redesign both the appearance and the structure of this program.

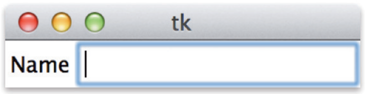
Laying Out the Widgets

One of the things that makes the color picker GUI ugly is the fact that everything is arranged top to bottom. tkinter uses this layout by default, but we can usually come up with something better.

To see how, let's revisit the example from *Getting Information from the User with the Entry Type, on page 322*, placing the label and button horizontally. We tell tkinter to do this by providing a side argument to method pack:

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text='Name')
label.pack(side='left')
entry = tkinter.Entry(frame)
entry.pack(side='left')

window.mainloop()
```

Here is the resulting GUI:



Setting side to "left" tells tkinter that the leftmost part of the label is to be placed next to the left edge of the frame, and then the leftmost part of the entry field is placed next to the right edge of the label—in short, that widgets are to be packed using their left edges. We could equally well pack to the right, top, or bottom edges, or we could mix packings (though that can quickly become confusing).

For even more control of our window layout, we can use a different layout manager called `grid`. As its name implies, it treats windows and frames as grids of rows and columns. To add the widget to the window, we call `grid` instead of `pack`. Do not call both on the same widget; they conflict with each other. The `grid` call can take several parameters, as shown in Table 27, *grid() Parameters*

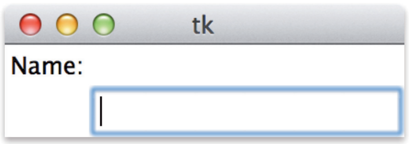
Parameter	Description
`row`	The number of the row to insert the widget into—row numbers begin at 0.
`column`	The number of the column to insert the widget into—column numbers begin at 0.
`rowspan`	The number of rows the widget occupies—the default number is 1.
`columnspan`	The number of columns the widget occupies—the default number is 1.

Table 27—grid() Parameters

In the following code, we place the label in the upper left (row 0, column 0) and the entry field in the lower right (row 1, column 1).

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text='Name:')
label.grid(row=0, column=0)
entry = tkinter.Entry(frame)
entry.grid(row=1, column=1)

window.mainloop()
```

Here is the resulting GUI; as you can see, this leaves the bottom-left and upper-right corners empty:



16.5 Introducing a Few More Widgets

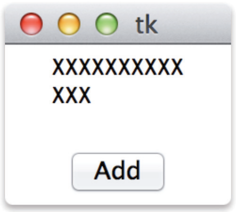
To end this chapter, we will look at a few more commonly used widgets.

Using Text

The Entry widget that we have been using since the start of this chapter allows for only a single line of text. If we want multiple lines of text, we use the Text widget instead, as shown here:

```
import tkinter

def cross(text):
    text.insert(tkinter.INSERT, 'X')

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()

text = tkinter.Text(frame, height=3, width=10)
text.pack()

button = tkinter.Button(frame, text='Add', command=lambda: cross(text))
button.pack()

window.mainloop()
```

Here is the resulting GUI:



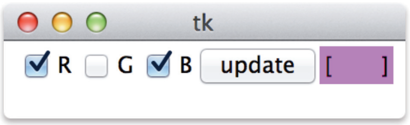
Text provides a much richer set of methods than the other widgets we have seen so far. We can embed images in the text area, put in tags, select particular lines, and so on. The exercises will give you a chance to explore its capabilities.

Using Checkbuttons

Checkbuttons, often called *checkboxes*, have two states: on and off. When a user clicks a checkbutton, the state changes. We use a tkinter mutable variable to keep track of the user's selection. Typically, an IntVar variable is used, and the values 1 and 0 indicate on and off, respectively. In the following code, we use three checkbuttons to create a simpler color picker, and we use method config to change the configuration of a widget after it has been created:

```
import tkinter

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
red = tkinter.IntVar()
green = tkinter.IntVar()
blue = tkinter.IntVar()

for (name, var) in (('R', red), ('G', green), ('B', blue)):
    check = tkinter.Checkbutton(frame, text=name, variable=var)
    check.pack(side='left')

def recolor(widget, r, g, b):
    color = '#'
    for var in (r, g, b):
        color += 'FF' if var.get() else '00'
    widget.config(bg=color)

label = tkinter.Label(frame, text='[       ]')
button = tkinter.Button(frame, text='update',
                        command=lambda: recolor(label, red, green, blue))
button.pack(side='left')
label.pack(side='left')
window.mainloop()
```

Here is the resulting GUI:



Using Menu

The last widget we will look at is Menu.

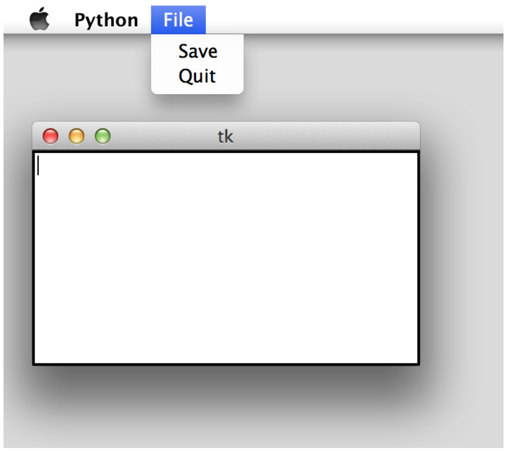
The following code uses this to create a simple text editor:

```
import tkinter
import tkinter.filedialog as dialog

def save(root, text):
  data = text.get('0.0', tkinter.END)
  filename = dialog.asksaveasfilename(
      parent=root,
      filetypes=[('Text', '*.txt')],
        title='Save as...')
    writer = open(filename, 'w')
    writer.write(data)
    writer.close()

def quit(root):
    root.destroy()

window = tkinter.Tk()
text = tkinter.Text(window)
text.pack()

menubar = tkinter.Menu(window)
filemenu = tkinter.Menu(menubar)
filemenu.add_command(label='Save', command=lambda : save(window, text))
filemenu.add_command(label='Quit', command=lambda : quit(window))

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

window.mainloop()
```

The program begins by defining two functions: save, which saves the contents of a text widget, and quit, which closes the application. Function save uses tkFileDialog to create a standard “Save as...” dialog box, which will prompt the user for the name of a text file.

After creating and packing the Text widget, the program creates a menu bar, which is the horizontal bar into which we can put one or more menus. It then creates a File menu and adds two menu items to it called Save and Quit. We then add the File menu to the menu bar and run mainloop.

Here is the resulting GUI:



16.6 Object-Oriented GUIs

The GUIs we have built so far have not been particularly well structured. Most of the code to construct them has not been modularized in functions, and they have relied on global variables. We can get away with this for very small examples, but if we try to build larger applications this way, they will be difficult to understand and debug.

For this reason, almost all real GUIs are built using classes and objects that tie models, views, and controllers together in one tidy package. In the counter shown next, for example, the application's model is a member variable of class `Counter`, accessed using `self.state`, and its controllers are the methods `up_click` and `quit_click`.

```
import tkinter

class Counter:
    """A simple counter GUI using object-oriented programming."""
    def __init__(self, parent):

        """Create the GUI."""

        # Framework.
        self.parent = parent
        self.frame = tkinter.Frame(parent)
        self.frame.pack()

        # Model.
        self.state = tkinter.IntVar()
        self.state.set(1)

        # Label displaying current state.
        self.label = tkinter.Label(self.frame, textvariable=self.state)
        self.label.pack()

        # Buttons to control application.
        self.up = tkinter.Button(self.frame, text='up', command=self.up_click)
        self.up.pack(side='left')

        self.right = tkinter.Button(self.frame, text='quit',
                                    command=self.quit_click)
        self.right.pack(side='left')

    def up_click(self):
        """Handle click on 'up' button."""

        self.state.set(self.state.get() + 1)
    def quit_click(self):
        """Handle click on 'quit' button."""

        self.parent.destroy()
if __name__ == '__main__':

    window = tkinter.Tk()
    myapp = Counter(window)
    window.mainloop()
```

16.7 Keeping the Concepts from Being a GUI Mess

In this chapter, you learned the following:

- Most modern programs provide a graphical user interface (GUI) for displaying information and interacting with users. GUIs are built out of widgets, such as buttons, sliders, and text panels; all modern programming languages provide at least one GUI toolkit.
- Unlike command-line programs, GUI applications are usually event-driven. In other words, they react to events such as keystrokes and mouse clicks when and as they occur.
- Experience shows that GUIs should be built using the model-view-controller pattern. The model is the data being manipulated; the view displays the current state of the data and gathers input from the user, while the controller decides what to do next.
- Lambda expressions create functions that have no names. These are often used to define the actions that widgets should take when users provide input, without requiring global variables.
- Designing usable GUIs is as challenging a craft as designing software. Being good at the latter doesn't guarantee that you can do the former, but dozens of good books can help you get started.

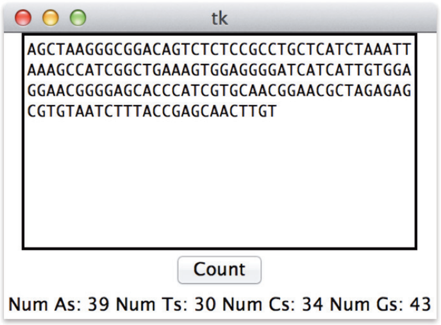
16.8 Exercises

Here are some exercises for you to try on your own. Solutions are available at

1. Write a GUI application with a button labeled "Goodbye." When the button is clicked, the window closes.
2. Write a GUI application with a single button. Initially the button is labeled 0, but each time it is clicked, the value on the button increases by 1.
3. What is a more readable way to write the following?

```
x = lambda: y
```

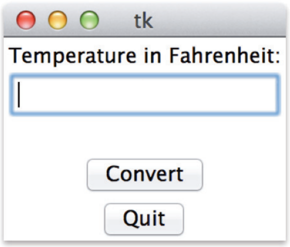
4. A DNA sequence is a string made up of *A*s, *T*s, *C*s, and *G*s. Write a GUI application in which a DNA sequence is entered, and when the Count button is clicked, the number of *A*s, *T*s, *C*s, and *G*s are counted and displayed in the window (see the image below).



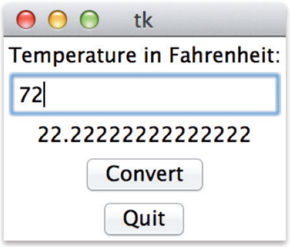
```
AGCTAAGGGCGGACAGTCTCTCCGCCTGCTCATCTAAATT
AAAGCCATCGGCTGAAAGTGGAGGGGATCATCATTGTGGA
GGAACGGGGAGCACCCATCGTGCAACGGAACGCTAGAGAG
CGTGTAATCTTTACCGAGCAACTTGT
```

```
Num As: 39 Num Ts: 30 Num Cs: 34 Num Gs: 43
```

5. In Section 3.3, *Defining Our Own Functions*, on page 35, we wrote a function to convert degrees Fahrenheit to degrees Celsius. Write a GUI application that looks like the image below.



When a value is entered in the text field and the Convert button is clicked, the value should be converted from Fahrenheit to Celsius and displayed in the window, as shown in the image below.



```
22.2222222222222
```

6. Rewrite the text editor code from *Using Menu*, on page 333, as an object-oriented GUI.