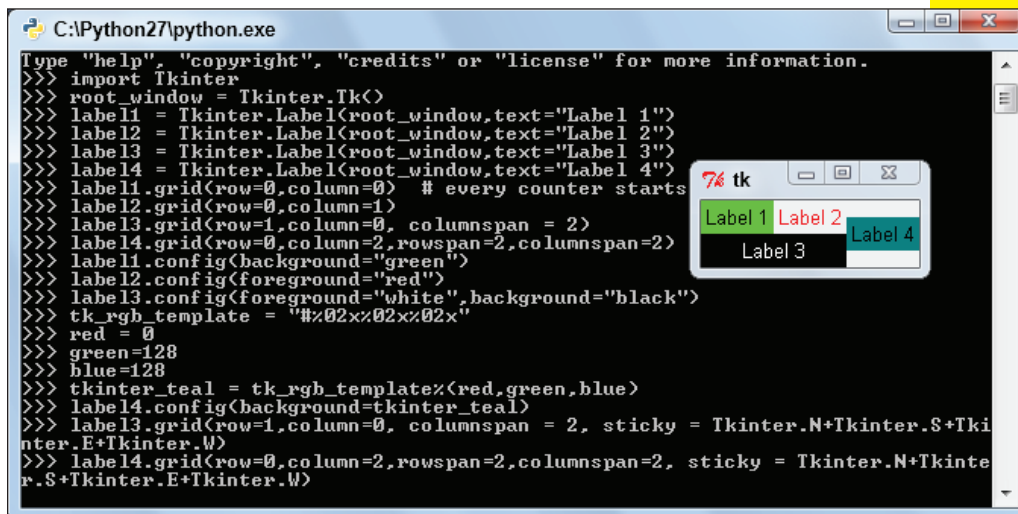


Hello GUI World!

Python's shell isn't the most beautiful thing in the world. In this project, you're going to rock a *graphical user interface (GUI)* with a toolkit called `Tkinter`. The toolkit has pretty much everything you could want from a GUI interface (even if some people think it could be prettier). The skills you use with `Tkinter` are skills you can use with other graphical user interface toolkits.

In this project you tackle putting together a basic program with a GUI, including opening windows, making and moving buttons and labels, and making the computer do stuff when the user clicks a button.



```
C:\Python27\python.exe
Type "help", "copyright", "credits" or "license" for more information.
>>> import Tkinter
>>> root_window = Tkinter.Tk()
>>> label1 = Tkinter.Label(root_window, text="Label 1")
>>> label2 = Tkinter.Label(root_window, text="Label 2")
>>> label3 = Tkinter.Label(root_window, text="Label 3")
>>> label4 = Tkinter.Label(root_window, text="Label 4")
>>> label1.grid(row=0, column=0) # every counter starts
>>> label2.grid(row=0, column=1)
>>> label3.grid(row=1, column=0, columnspan = 2)
>>> label4.grid(row=0, column=2, rowspan=2, columnspan=2)
>>> label1.config(background="green")
>>> label2.config(foreground="red")
>>> label3.config(foreground="white", background="black")
>>> tk_rgb_template = "#%02x%02x%02x"
>>> red = 0
>>> green=128
>>> blue=128
>>> tkinter_teal = tk_rgb_template%(red,green,blue)
>>> label4.config(background=tkinter_teal)
>>> label3.grid(row=1, column=0, columnspan = 2, sticky = Tkinter.N+Tkinter.S+Tkinter.E+Tkinter.W)
>>> label4.grid(row=0, column=2, rowspan=2, columnspan=2, sticky = Tkinter.N+Tkinter.S+Tkinter.E+Tkinter.W)
```

The screenshot shows a Python shell window with the following code and a small GUI window titled 'tk'. The GUI window contains four labels: 'Label 1' (green background), 'Label 2' (red foreground), 'Label 3' (white foreground, black background, spanning two columns), and 'Label 4' (teal background, spanning two rows and two columns).

Make a Quick Hello GUI World!



You need to use Python (command line) for the code with the interactive prompt `>>>` in this project and *not* the IDLE Shell window. If you use IDLE, you won't get the same effects. IDLE uses Tkinter itself, and this affects how your code works in the Shell.

1. Fire up Python (command line).

2. Type in the following code:

```
>>> import Tkinter
>>> label_widget = Tkinter.Label(None, text="Hello GUI World!")
```

You should see a new open window, with a title `'tk'`, like the one in Figure 11-1. The window is empty and small.

3. Type in this code and watch your tk window:

```
>>> label_widget.pack()
```

<voiceover>Achievement unlocked: Hello World in GUI
</voiceover>

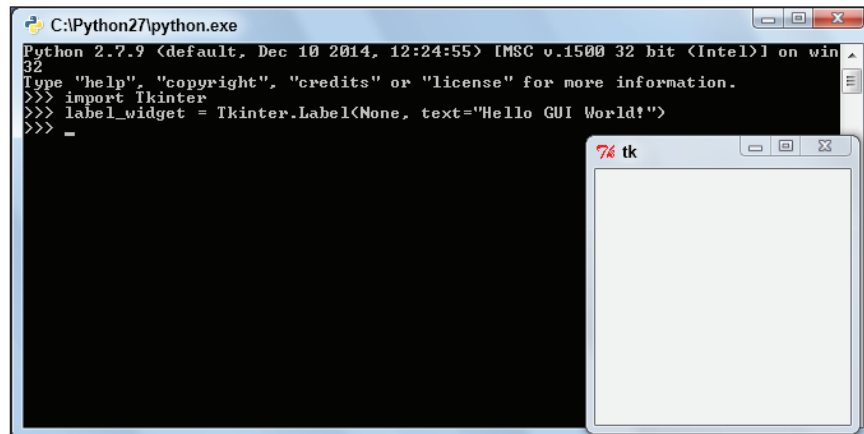


Figure 11-1: The default window is open for you.



That's tee-kay-inter

Tkinter is a Python interface for a GUI toolkit called Tk. So, you pronounce it tee-kay-inter. During a conference, I mispronounced it ta-kinter. So embarrassing!

Want to know more than just how to pronounce it? Look at effbot.org (<http://effbot.org/tkinterbook>) and New Mexico Tech's Tkinter reference (<http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html>).

Two things are probably happening right now:

- ✓ You noticed that your tk window changed from something small (but still noticeable) into something teeny tiny and barely big enough to fit text into.
- ✓ You've gone weak at the knees because you just wrote your first GUI program in only three lines of Python code. If you need to lie down, go ahead. When you've recovered from the shock, see if you can do this in *two* lines of code.

Analyze Hello GUI World!

There's not a lot to this program, but you should be able to work out the following on your own:

- ✓ Tkinter is a module. That's sort of easy to see because if you use `import` on something then it's a module.
- ✓ `Tkinter.Label` is an attribute of the Tkinter module. It's used with parentheses, so this means it's either a function or a class.

Ah ha! It has a capital letter, so it should be a class. When you use classes with parentheses you instantiate the class — that is, you make an instance of that class.

- ✓ `label_widget` has its own method called `pack()`. This means `label_widget` is probably a custom object. That supports the theory that `Tkinter.Label` is a class.

The variable `label_widget` really does store an instance of a `label` widget.



Widgets are pieces of code commonly used in GUIs. For example, GUIs often show text in some area but don't let the user change that text. The `label` widget lets you do that. It makes it easier for you to create different kinds of applications using the same or similar code.

Each and every time you use a graphical interface on a computer, you're using widgets. They're unavoidable. You're so familiar with them that you don't notice them, just like you don't notice the wallpaper.

You've met one widget here — a `label` widget. You use a `label` widget to show the user text or images when the user is not supposed to be able to change those things.

In the window you just created, you can see a lot of widgets:

- ✓ A menu button in the top-left corner
- ✓ A title (`tk`)
- ✓ Three buttons: minimize, maximize, and close

You might not be able to see some of these because the window is so small. Make it larger to see them all, like in Figure 11-2. You could probably also call the bars at each side of the window widgets.

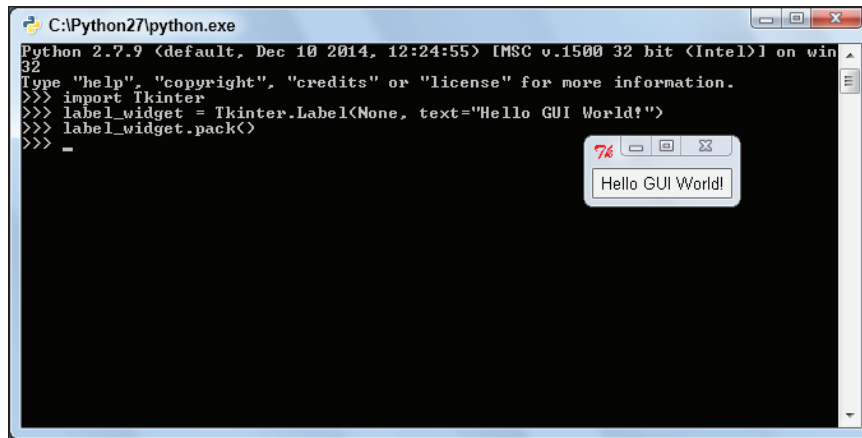


Figure 11-2: Your Hello GUI World! window already has widgets. The OS gave them to you.

Widgets usually look different on different operating systems — Windows, Macintosh, Linux. But their main job is pretty much the same no matter what. That means what you learn here while you’re working with `Tkinter` works on other widget sets.



- ✔ The fact that you can’t see a widget until you call its `pack` method comes back to bite you on the backside again and again during your programming adventures, so listen up! You’ll get bitten because you’ll think that you’ve added a widget — but you won’t be able to see it because you haven’t packed it. *Remember* that you can’t see your widgets until you pack them (or, when you read about it later, `grid` them).
- ✔ The escapade with `pack` means that the widget exists before you call `pack`. The widget exists whether you can see it or not. When I talk about widgets, I don’t just mean the way the widget looks on the screen.
- ✔ The window you created hangs around and you’re free to keep programming in the Shell window. Try `print("I can still code here!")`. In other programs, you can’t do any more coding when Python was doing something.



Make sure you call the `pack` method of each of your widgets or you can't see them!

Run Hello GUI World! from a File

Now try to run Hello GUI World! from a script saved in a Python file. You can use the IDLE Editor window for this:

1. Create a new file called `hello_gui_world.py`.
2. Type the following code into the file (the earlier code with the `>>>` prompts stripped out):

```
import Tkinter
label_widget = Tkinter.Label(None, text="Hello GUI World!")
label_widget.pack()
```

3. Save it and run it.

You get a restart and . . . nothing.

4. Add the following line at the end:

```
Tkinter.mainloop()
```

Now when you save and run it, you get your window with the little widget back. Hurrah! The real thing here is `Tkinter.mainloop()`.

The `mainloop` function is an infinite loop. Without it, `Tkinter` won't work. When you type your `Tkinter` code in a Shell, it's supposed to be interactive, so Python gets the `mainloop` automatically going for you.



Click the `close` widget (the X in the top-right corner of the title bar) to close the program.

Understanding `mainloop`

`Tkinter.mainloop` is the reason that the window can stay there and you can keep typing in the Shell window. It lets more than one thing happen at once, so you don't get blocked.



`Tkinter.mainloop` takes control, constantly checking for events and, if needed, sending those events to the program. An *event* is anything to do with the program or interface, but most of the time an event is either a keystroke or a mouse action. Like when the user presses Q to quit, for example.

Sometimes what you think is one event is actually more than one. (For example, a keystroke can be two separate events — a key down and a key up.) Your Hello GUI World! processes one event in particular — a mouse click on the close button. This, not surprisingly, causes the application to close.

In *pseudo code* — which isn't actual code, but an English description disguised as code — `Tkinter.mainloop` does something like this:

```
def mainloop():
    while True:
        event = wait_for_event()
        if event == close_button: # someone clicked the
            close button...
                clean_up_and_shut_down()
                break
        dispatch_event_to_the_relevant_handlers(event)
```

Basically, when you run `mainloop()`, `Tkinter` takes control of your program! You can see from the pseudo code that the only time it leaves `mainloop` is when it's cleaning up and shutting down or dispatching an event.

To *dispatch an event* means that `Tkinter` calls a function and passes (a variable holding) the event as an argument to the function. (*Dispatch* in plain English means to send something quickly.)



In practice, events tend to be sent to one of your methods (a function of an instance of a class) rather than a function at the module level. I use the word *method* from here on.

Once `mainloop` starts, you've got a short chance to communicate with Tkinter. Indeed, the only time any part of your program runs after `mainloop` has started is when `mainloop` has called one of your program's methods. All bow before the mighty `mainloop`!

You pass control over to `mainloop` because `mainloop` has some code to check for events without holding up the rest of the program.

This way of making a program flow is a lot different from the programming you did in the book projects. In those, you had a reasonably good idea of how the program would flow through your code and what functions would be called in what order. But now your code only does something when `mainloop` responds to an event that it receives. Because the program's flow is driven by the events, this form of programming is called *event-driven programming*.

Make Sure Tkinter Calls You Back



Because you can't do anything once `Tkinter.mainloop` takes control, you have to make sure that Tkinter knows how and when to get back in touch with you before that happens. This means setting up Tkinter with all it needs to know before you call `Tkinter.mainloop`.

Arranging for Tkinter to call you back is *registering a callback*. (Who'da thunk it?) To register a callback, give Tkinter

- ✓ The name of the event you want it to respond to. Sample events are `<Button-1>` (pressing the left mouse button) and `<Key>` (pressing a key on the keyboard).

- ✓ The name of the method that you want Tkinter to call you back on. Usually you make these names yourself. The first one you make later is called `button_callback`.

The method that you register, since it handles the callback that Tkinter makes, is also called a *callback* (not entirely unambiguous, I know).



Don't forget. When you're writing an application, you need to

- ✓ Think about what events — like mouse clicks and keyboard strokes — you want your program to deal with.
- ✓ Write separate methods to deal with each event (or categories of event).
- ✓ Register each of these methods with Tkinter. The events are typically connected with a widget, so you tend to register the callback at the time you make the widget instance.
- ✓ Start `mainloop`, then wait for Tkinter to call your callback back.

Time to get your hands dirty.



Don't be afraid of Doug

A phrase like “not entirely unambiguous” is a rhetorical device called *litotes* (*lie-tote-eeze*). It uses a negative statement to reinforce a positive. In the text, the word *callback* describes both the process of calling back and the thing that's called back. In reality, it *is* pretty ambiguous.

Monty Python have a skit about two gangsters called the Piranha Brothers. One of those brothers, Doug, is a vicious, terrifying brute. (Grown men have apparently torn their own heads off to avoid facing Doug.) Doug is terrifying because of his knowledge of rhetoric (including litotes, satire, and metaphor). Doug was apparently ignorant of *synecdoche*. Look it up.

Add a Clickable Button

You can't register a callback on a label (possibly because labels don't do anything), but you can register a callback on a kind of widget called a `button`. Do it in two stages. First make the button:

1. **Make a new file called `tkinter_button.py`.**
2. **Import the `Tkinter` module.**

```
import Tkinter
```

3. **Create an instance of `Tkinter.Button` and use the `text` config option equal to `"Click me!"`**

It's the same as when you created a `Label`. Just replace `Label` with `Button`:

```
button_widget = Tkinter.Button(None, text="Click me!")
```

4. **Pack the button: `button_widget.pack()`.**
5. **Call `mainloop`: `Tkinter.mainloop()`.**

There aren't any callbacks at the moment but when you run it, you get a button that asks you to click it. This is the program:

```
import Tkinter
button_widget = Tkinter.Button(None, text="Click me!")
button_widget.pack()
Tkinter.mainloop()
```

When I run it, a teeny tiny window opens up. It's kind of hard to find, so make sure you look for it. When you do, click it. It looks like it pops in and out as you click it. This is the button as it *toggles* (changes) from selected to deselected. Once you notice that, you can close the window by clicking the `close` widget.

Now for stage two: adding the callback.

- 6. Create a function after the import of Tkinter, but before button_widget is instantiated. Call it button_callback.**

It takes no arguments.

```
def button_callback():
```

- 7. In the function, print something.**

```
    print("Click me again!")
```

- 8. Go back to the line where you created button_widget and add command=button_callback as another argument to the Tkinter.Button.**

This code is on two separate lines. You can type it as is, if you indent the start of the second line to be under the None in the first line (or you can try to fit it all on a single line).

```
    button_widget = Tkinter.Button(None, text="Click me!",  
                                   command=button_callback)
```

This is the code. The callback is pretty simple:

```
import Tkinter  
def button_callback():  
    print("Click me again!")  
button_widget = Tkinter.Button(None, text="Click me!",  
                                command=button_callback)  
  
button_widget.pack()  
Tkinter.mainloop()
```

Run the code, find that tiny window, and click that button. You'll see Click me again! print in IDLE's Shell window each time you click the button. See Figure 11-3.

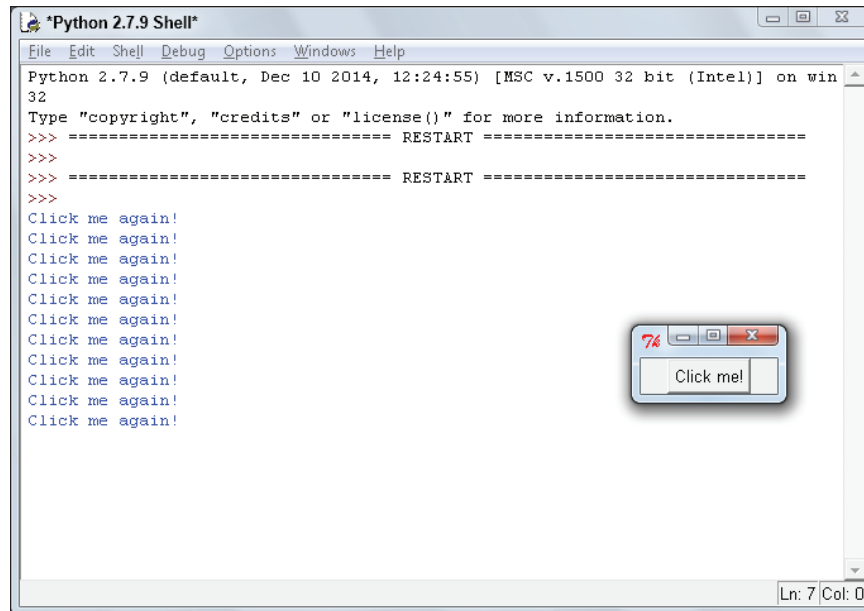


Figure 11-3: This button widget is hooked up to a callback.

Change the Button to a Quit Button



You need to specifically hook up your callbacks. It isn't good enough to create a callback that matches what you think the callback should do.

To demonstrate this, you're going to make a quit button and corresponding quit code:

1. Create a new file called `quit_gui.py`.
2. Import the Tkinter module:

```
import Tkinter
```

3. Create a function called `quit`, which takes no arguments. Put it at the top of the file, just after the `import` statement.

Put it at the top so you define your functions before you call them.



It's convention to put imports at the very top of the file so that they can be found easily. At the moment, the function can be only a `print` statement reminding you to fix it up later. If you want a challenge, go back to book for Project 5 and read up on how to use `sys.exit()`:

```
def quit():
    print("Need to quit from this function")
```

4. Create a button widget called `quit_widget` with the text "Quit".

```
quit_widget = Tkinter.Button(None, text="Quit")
```

5. Pack the widget.

```
quit_widget.pack()
```

6. Call `mainloop`:

```
Tkinter.mainloop()
```

This is what you should get:

```
import Tkinter
def quit():
    print("Need to quit from this function")

quit_widget = Tkinter.Button(None, text="Quit")
quit_widget.pack()
Tkinter.mainloop()
```

7. Run the code!

You should get a window that looks like Figure 11-4.

The `quit` function isn't called automatically. This is because you haven't registered the function with `Tkinter` before `mainloop` was called. This may seem obvious, but it's not enough to have a function called `quit` (or whatever).

`Tkinter` isn't smart enough to work out that you want the function connected with the button. You need to tell it you want the `quit` function to run when the button is pressed by using the `command=quit` argument.

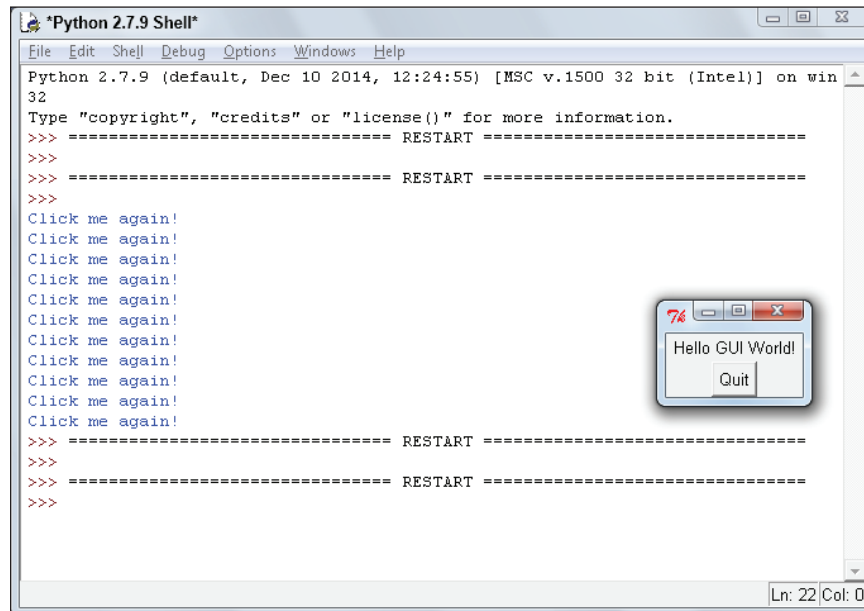


Figure 11-4: Clicking does nothing. Need to expressly hook up the callback.

8. When you're satisfied that Tkinter can't work out how to link up callbacks, you can hook up the callback explicitly:
9. Change the button code to `quit_button = Tkinter.Button(None, text="Quit", command=quit).`

There's a standard way to exit a Tkinter application, but you need to know a few more things before I can show you.

Changing Button Options with `config`



The arguments you pass by keyword when you're instantiating a Tkinter widget are called *options*. You register the callback at the time of instantiation by passing the callback to the `command` option. That means, make sure you've got an option `command=<callback name>` like in the code `Tkinter.Button(None, text="Click me!", command=button_callback)` in the following example.



If for some reason you don't or can't set an option when you instantiate a widget, you can use the widget instance's `config` method to set that option at a later time:

```
import Tkinter
def button_callback():
    print("Click me again!")
button_widget = Tkinter.Button(None, text="Click me!")
button_widget.pack()
button_widget.config(command=button_callback)
Tkinter.mainloop()
```

This code works in exactly the same way as the earlier code, even though registering the callback is the last thing that you do (even after packing the widget).

You can use the `config` method to change most aspects of your user interface while the program is running. For example, you could change the color of a button from green to red to show the user that time is running out.



If you can set an option when you instantiate the widget, then you can change that option using the `config` method. Sometimes you can't (at least not easily) set an option when the widget is created. The `config` method allows you to create the widget and then set the option later when you have the info you need.

Here's an example using it to replace an existing callback with a new one. You'd do this if you want to have a button do different things (like you change the user interface halfway through your application and reuse a button that you've already made).

```
import Tkinter
def button_callback():
    print("Click me again!")
def button_callback2():
    print("Ok, I've had enough of clicking now")
```

```
button_widget = Tkinter.Button(None, text="Click me!",
                                command=button_callback)

button_widget.pack()

button_widget.config(command=button_callback2)

Tkinter.mainloop()
```

In this code, the callback `button_callback` is initially assigned, but later `config` is used to change the registration. The function `button_callback` is never called (even though it's initially assigned), only `button_callback2` is called.

You can also change callbacks midflight, so to speak. While it's better to have all your callbacks sorted before you call `mainloop`, you can assign or change callbacks afterwards — but only from within a callback that Tkinter actually calls.

```
import Tkinter

def button_callback():
    print("Click me again!")
    button_widget.config(command=button_callback2)

def button_callback2():
    print("Ok, I've had enough of clicking now")

button_widget = Tkinter.Button(None, text="Click me!",
                                command=button_callback)

button_widget.pack()

Tkinter.mainloop()
```

Here, at the time the button is first clicked, `button_callback` is registered with Tkinter. While `button_callback` is being processed, the registration of `button_widget` is changed from `button_callback` to `button_callback2` (using `button_widget`'s `config` method). After that happens, `button_callback` is never called again. All later clicks go to `button_callback2`.

Create a Root Window



When you instantiate the widgets in the label example, a window popped up that you didn't ask for. That window is called the *root widget* or *root window*. It's called the root widget because programmers think of all the widgets that they create as growing on top of one another. If one widget has another, the first widget is a *parent* (or *parent widget*) of the second. The second widget is said to be a *child* (or *child widget*) of the first.

Python created that window because it knew you'd need it. (Opening a root window is also the way that Tkinter is initialized.) Letting this happen is slightly poor form because you're not being *explicit* about what you want the program to do — you're not telling it exactly what to do.

Instead, you're relying on a particular behavior of Tkinter. When you cause something to happen without giving an express instruction for it you're *implicitly* doing that something.



It's generally better to be explicit. When you're explicit, the code tells everyone else what is happening and why.

If you explicitly create a root window then you can save a reference to it and work with that reference. (You can recover a reference to the root window from any widget attached to it. Use `dir` on the widget and work out which attribute is the right one.)

You create a root window by using `Tkinter.Tk()`. This function initializes a root window and returns a reference to it. You want to catch this reference, so assign its return value to a variable.

Here's Hello GUI World! rewritten with an explicit root window. Note that the instantiation of the label now *explicitly* references the root window as its first option:

```
>>> import Tkinter
>>> root_window = Tkinter.Tk()
```

The default window appears at this point (and not when the label is instantiated, like in the earlier example). You can see it in Figure 11-5.

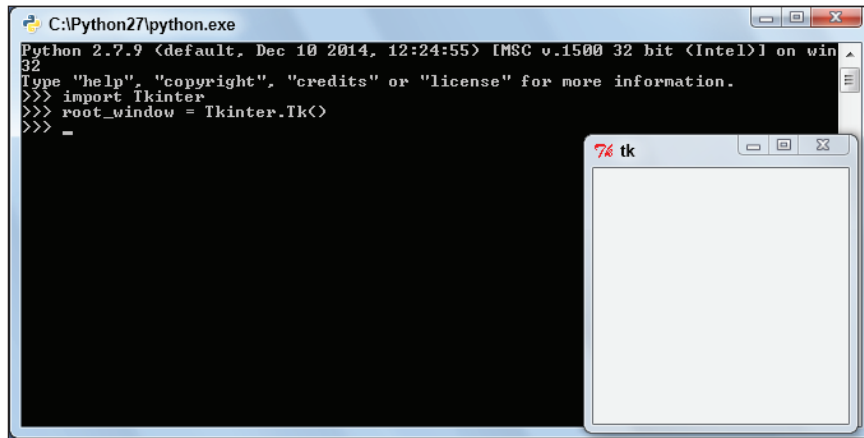


Figure 11-5: Opening a parent, or root, window explicitly.

```
>>> label_widget = Tkinter.Label(root_window,
                                   text="Hello GUI World!")
>>> label_widget.pack()
```

From now on in the project, you should explicitly create a root window. You got this!

After you have a reference to the root (or any other) window, you can change the window's width, height, and location by using its *geometry* method. The *geometry* method needs to receive a string as its argument. It needs to be in the form *widthxheight+xoffset+yoffset*. That's width by height, with a horizontal (x) offset and a vertical (y) offset and all of them as numbers. The unit is pixels. You read more about pixels a little later. (There's so much to cover!)

For now, when you run the following code, notice that the size and location of the root window changes after you enter the last line:

```
>>> import Tkinter
>>> root_window = Tkinter.Tk()
```

```
>>> label_widget = Tkinter.Label(root_window,
                                text="Hello GUI World!")
>>> label_widget.pack()
>>> new_geometry_template= "%dx%d%+d%+d"
>>> new_geometry = new_geometry_template%(200,150,100,100)
>>> new_geometry
'200x150+100+100'
>>> root_window.geometry(new_geometry)
```

Note that `new_geometry` is a string. When the new geometry is applied, you should see the window get bigger and change position. This change isn't animated; it happens instantaneously.

Since everything in Python is an object, it's reasonable to assume that widgets are objects as well. Widgets are pretty complex, as far as objects go. Use `dir(label_widget)` to get a list of its attributes. There are almost 200 of them. The attributes can give you a lot of information that'll help you.

Quit the Tkinter Way

To close a Tkinter application, you're supposed to call the `destroy` method on its root window. Now that you know how to get an explicit reference to the root window, you have the power to destroy it:

- 1. Open your `hello_gui_world.py` program.**
- 2. Add a line explicitly creating a root window.**

```
root_window = Tkinter.Tk()
```

- 3. Make the root window the parent of each widget you create.**

For example, replace `None` by the variable referencing the root widget in the widget constructors.

4. Change the `quit()` function to call the `destroy()` method of the root window. It doesn't take any arguments.

```
def quit():  
    root_window.destroy()
```

Here's the code:

```
import Tkinter  
  
def quit():  
    root_window.destroy()  
  
root_window = Tkinter.Tk()  
quit_button = Tkinter.Button(root_window, text="Quit",  
                             command=quit)  
  
quit_button.pack()  
Tkinter.mainloop()
```

The `quit` function can see the variable `root_window`, so it can call all its methods.

Use Two or More Widgets at the Same Time

When you have more than a handful of widgets, you face a new problem — how do you arrange the widgets in the parent window and in relation to each other?



How you arrange widgets is called the window's *geometry* and it's managed by a *geometry manager*. I'm going to introduce you to two Tkinter geometry managers — `pack` and `grid`. You'll use either `pack` or `grid` whenever you need lots of widgets in your application — that is almost always.

Default widget packing

Type the following in your Python (command line) shell to get instant feedback about how the widgets are being arranged:

```
>>> import Tkinter
>>> root_window = Tkinter.Tk()
>>> label1 = Tkinter.Label(root_window, text="Label 1")
>>> label2 = Tkinter.Label(root_window, text="Label 2")
>>> label1.pack()
```

The first label should appear now.

Now type this:

```
>>> label2.pack()
```

The second label appears. It's underneath the first one. See Figure 11-6.

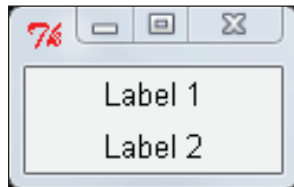


Figure 11-6: The second label appears underneath the first label.

Pack a widget on the bottom or side of a window

As you add widgets, they're packed underneath the widgets that are already there. That's nice if that's where you want them, but not so nice if you want them somewhere else.

To see how to make widgets stick to different sides, you need to increase the size of `root_window`:

```
>>> new_geometry_template= "dx%d%+d%+d"
>>> new_geometry = new_geometry_template%(200,200,100,100)
>>> root_window.geometry(new_geometry)
```



You can make them hug a different wall by packing them and specifying the side. You can repack a widget (in a different place, for example) by just calling its `pack` method again. Make `label2` stick to the bottom by calling its `pack` method and specifying the side to be `Tkinter.BOTTOM`:

```
>>> label2.pack(side=Tkinter.BOTTOM)
```

The second label sticks to the bottom of the window. If you resize and move the window manually, `label1` sticks to the top and `label2` sticks to the bottom of the window.

You should be able to tell, since `Tkinter.BOTTOM` is in ALLCAPS, that it's a constant (and that it means put the widget at the bottom of its parent).



Options for placing your widgets are `BOTTOM`, `RIGHT`, `LEFT`, and (the default) `TOP`.

See Figure 11-7 and try adding some more widgets using left and right packing:

```
>>> label3 = Tkinter.Label(root_window, text="Label 3")
>>> label3.pack(side=Tkinter.RIGHT)
>>> label4 = Tkinter.Label(root_window, text="Label 4")
>>> label4.pack(side=Tkinter.LEFT)
```

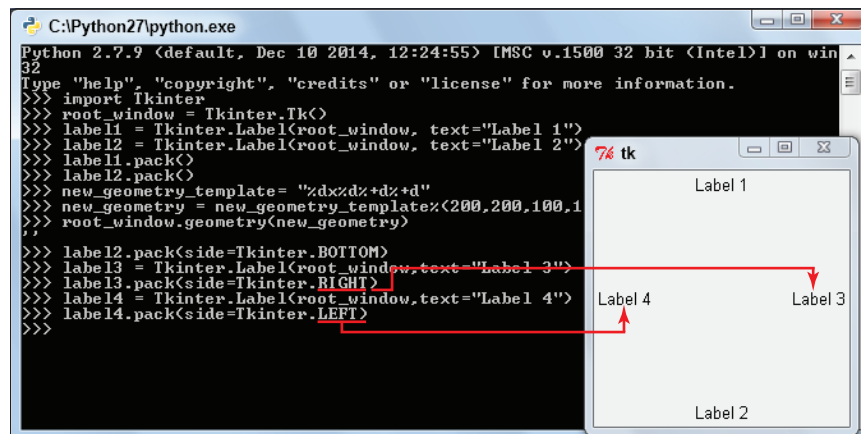


Figure 11-7: The third label is packed on the right; the fourth label is packed on the left.

Use a frame for a complex arrangement of widgets

Resize the root window and make sure the widgets stick to the side you packed them on. Guess what? There's no way to put a widget next to `label1`:

```
>>> label4.pack(side=Tkinter.TOP)
```

This code moves `label4` to the top, but it's offset a little to the left. See Figure 11-8. It's offset because of how `label3` was packed. You can solve most packing problems with one or more `Tkinter.Frame` widgets. *Frames* are invisible spaces where you can organize widgets as a group. You can pack the frames themselves into the windows.

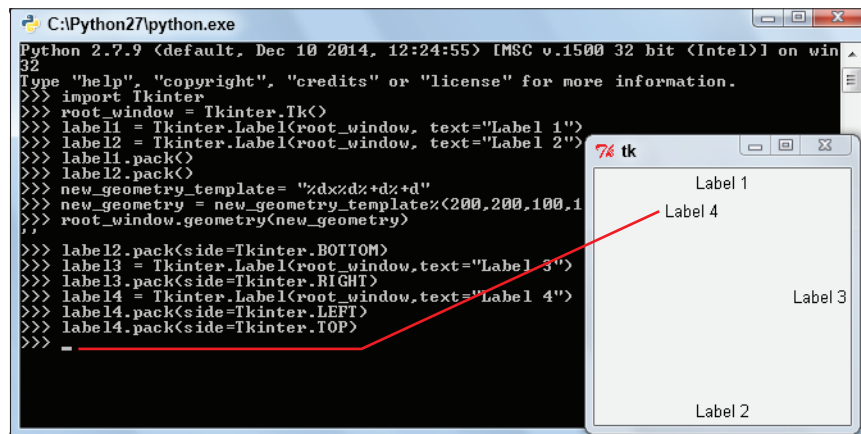


Figure 11-8: The fourth label is repacked at the top, but doesn't fit right.

The following section explains how to arrange your widgets using `Tkinter.Frame`s. For example, the following steps create three rows. The first row has three widgets, the second row has two widgets, and the bottom row has one widget.

Type the code in as you do each step:

1. Create your root window:

```
>>> import Tkinter
>>> root_window =Tkinter.Tk()
```

2. Create three Tkinter.Frame called row1, row2, and row3 with the root window as the parent of each:

```
>>> row1 = Tkinter.Frame(root_window)
>>> row2 = Tkinter.Frame(root_window)
>>> row3 = Tkinter.Frame(root_window)
```

3. Pack each of these Tkinter.Frame widgets in order:

```
>>> row1.pack()
>>> row2.pack()
>>> row3.pack()
```

You won't see anything spectacular here — the widgets are invisible, but they're there.

4. Create three Tkinter.Label widgets with row1 as their parent:

```
>>> label1 = Tkinter.Label(row1, text="Label 1")
>>> label2 = Tkinter.Label(row1, text="Label 2")
>>> label3 = Tkinter.Label(row1, text="Label 3")
```

5. Pack each of the widgets into frame row1. Use Tkinter.RIGHT.

```
>>> label1.pack(side=Tkinter.RIGHT)
>>> label2.pack(side=Tkinter.RIGHT)
>>> label3.pack(side=Tkinter.RIGHT)
```

Notice that the widgets appear from the right in the order they are packed.

6. Create two more Tkinter.Label widgets but use row2 as their parent. Pack them on the side Tkinter.LEFT.

```
>>> label4 = Tkinter.Label(row2, text="Label 4")
>>> label5 = Tkinter.Label(row2, text="Label 5")
```



```
>>> label4.pack(side=Tkinter.LEFT)
>>> label5.pack(side=Tkinter.LEFT)
```

7. Create a Tkinter.Label as your final widget, and use row3 as its parent. Pack the widget.

```
>>> label6 = Tkinter.Label(row3, text="Maybe this could be a
status bar?")
>>> label6.pack(side=Tkinter.BOTTOM)
```

When that’s all done, you should end up with something that looks like Figure 11-9. You can see that if you’re careful, you can create complicated layouts using Tkinter.Frame widgets to group other widgets together.

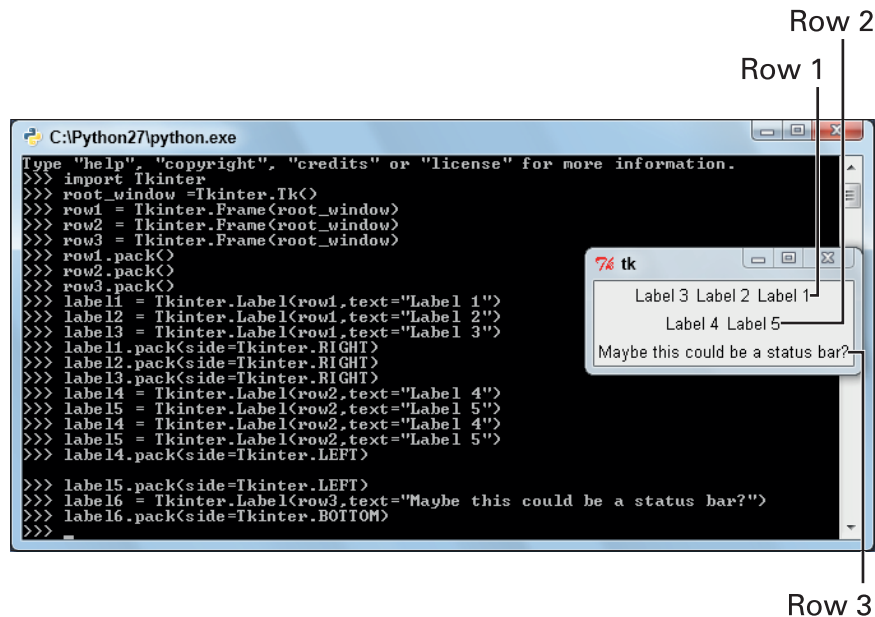


Figure 11-9: Carefully using frames lets you make complicated arrangements.

Laying widgets in a grid

Tkinter also has a grid geometry manager. For it, use the widget’s grid method, to say which row and column to place the widget. You can make the widget stretch more than one row or column.



In any root window, use either `pack` or `grid`, but not both. If you use `pack` on a widget, don't use `grid` for any other widget that has the same root window (and vice versa).

Type the following code to create an application with gridded widgets. You use the widget's `grid` method, setting a `row` and `column` option to locate the widget. Each `row` and `column` is a number 0 or greater. As you add widgets, the layout grows to fit them in:

```
>>> root_window = Tkinter.Tk()
>>> label1 = Tkinter.Label(root_window, text="Label 1")
>>> label2 = Tkinter.Label(root_window, text="Label 2")
>>> label3 = Tkinter.Label(root_window, text="Label 3")
>>> label4 = Tkinter.Label(root_window, text="Label 4")
>>> label1.grid(row=0, column=0) # every counter starts at
    zero!
>>> label2.grid(row=0, column=1)
>>> label3.grid(row=1, column=0, columnspan = 2)
>>> label4.grid(row=0, column=2, rowspan=2, columnspan=2)
```

The `grid` method assumes that there's a table with empty spots to plop the widgets into. The table starts empty, but grows as it needs to. See Figure 11-10. Take some time to see how the `row=` and `column=` entries affect the final placement of the widgets.

- ✓ *Rows* are cells going across the screen. The `row=` config option refers to the row number. The greater the number, the farther down the cell is.
- ✓ *Columns* are cells going down the screen. The `column=` config option refers to the column number. The greater the number, the farther to the right the cell is.
- ✓ The top-left corner of the table is `row=0, column=0`.
- ✓ The cell beneath it is `row=1, column=0`.
- ✓ The cell to the right of it is `row=0, column=1`.

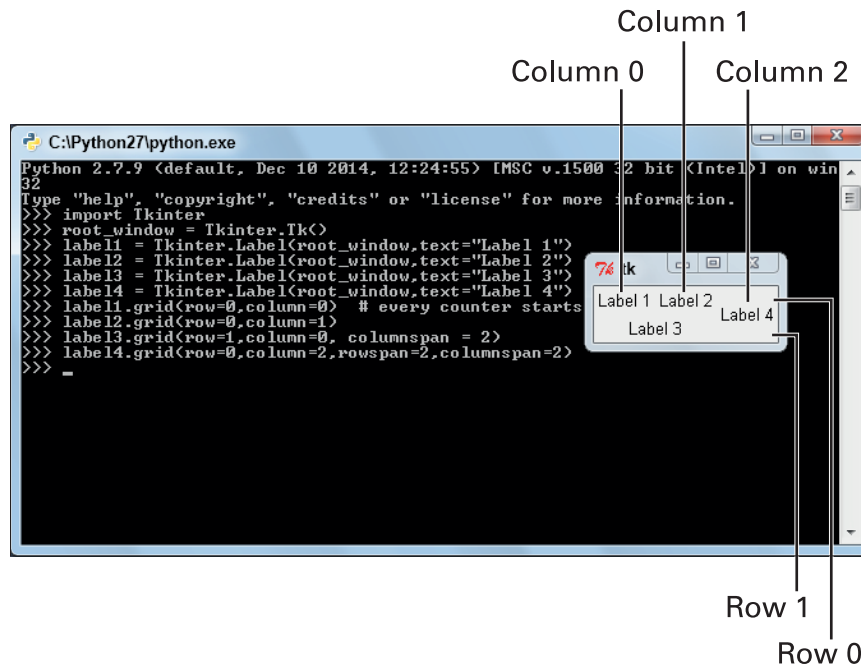


Figure 11-10: You can lay out widgets with the grid geometry manager.

With the `columnspan` and `rowspan` arguments, you can make widgets fill up more than one cell. For example, the widget `label3` has `columnspan=2`, so it takes up a single row, but stretches across two columns. The widget `label4` stretches across two rows (`rowspan=2`) and two columns (`columnspan=2`).

Don't close that window yet; you use it in the next section.

Color Your Widgets

To set foreground and background colors for your widgets, use the name of the color as a string: like "red", "green", and "blue". Like this:

```

>>> label1.config(background="green")
>>> label2.config(foreground="red")
>>> label3.config(foreground="white", background="black")
    
```

You can see them in all their colorful glory in Figure 11-11. If you hadn't changed the foreground color of `label3`, you wouldn't be able to read the text.

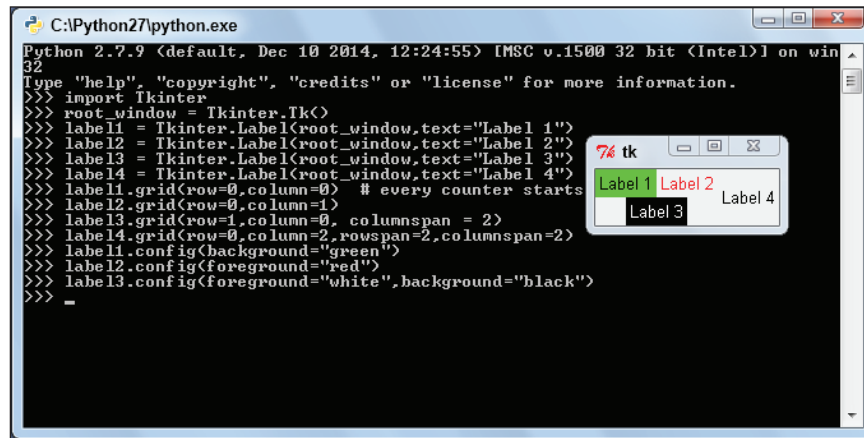


Figure 11-11: Widgets in color. Not all the space is filled.

For a label, the `background` option affects the color of the (you guessed it) background. The `foreground` option is for the widget's text.

Tkinter only knows certain colors, but you can make different colors by picking a number between 0 and 255 for red, green, and blue. But there's a catch . . . specifying them using hexadecimal notation! (Argh!)



You don't have to know what hexadecimal is. You just need to know that Python can take a number and convert for you. Use the `%x` format specifier in a string template to convert an integer into a hexadecimal number.

Use the template string `"#%02x%02x%02x"` to convert three integers from 0 to 255 (one each for red, green, and blue in that order) into a format that Tkinter can use.

If you know the number for red, green, and blue components — in that order — you can use this template to write your color. For

example, the color teal is a mixture of green and blue. Use this to make it:

```
>>> red = 0
>>> green = 128
>>> blue = 128
```

You pack these into a tuple and pass them to the template to give you a color that Tkinter can use:

```
>>> tk_rgb_template = "#%02x%02x%02x"
>>> tkinter_teal = tk_rgb_template%(red,green,blue)
>>> tkinter_teal
'#008080'
```

Now apply it to one of the widgets. See Figure 11-12.

```
>>> label4.config(background=tkinter_teal)
```



If you're stuck for colors, plenty of Internet sites give you values. Paint programs like Gimp also tell you the numbers of any color in that program.

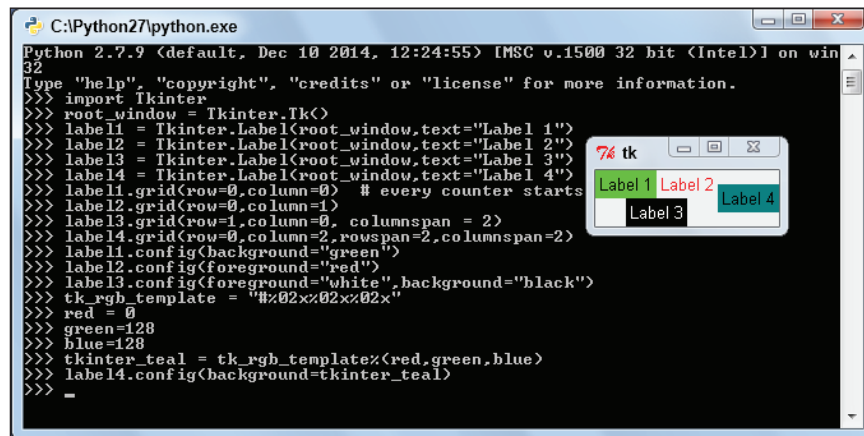


Figure 11-12: Teal is assigned to background color of widget 4.

Stretch Your Widgets

If you're using the grid geometry manager `Tkinter` assumes that the widget should be no bigger than it has to be. That's why the colored backgrounds in the `grid` example didn't have color all the way through. To have a widget fill the space it is in (the larger, parent widget that contains it), you need to do extra work. You do something different if you're using the `grid` geometry manager than when you are using the `pack` manager.

Stretch with the `grid` geometry

When you're assigning a row and a column, you can also assign what the widget sticks to. Then when the grid changes sides, the widget itself expands or stays the same, depending on what sides of the cell it's supposed to stick to.



Use the `sticky` option to set which cell walls the widget will stick to.

Using the window and colors from the previous section, you can change (for example) `label3` to stick to all the cell walls, like this:

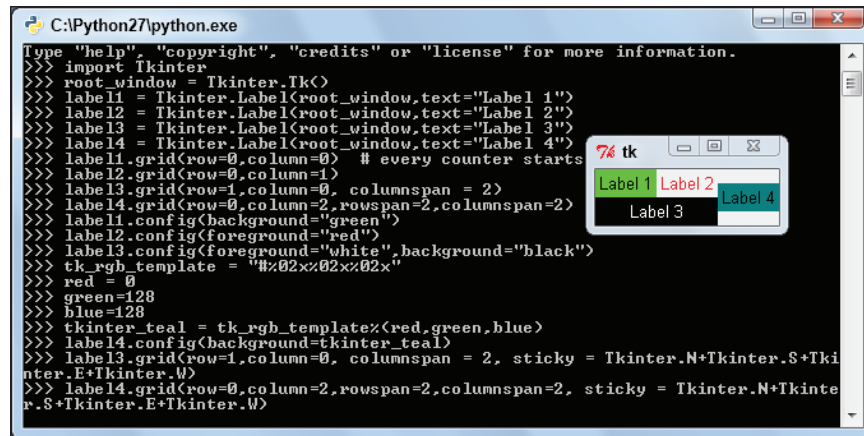
```
>>> label3.grid(row=1,column=0, columnspan = 2,
                sticky = Tkinter.N+Tkinter.S+Tkinter.E+Tkinter.W)
```

You can change `label4` like this. See Figure 11-13 as well:

```
>>> label4.grid(row=0,column=2, rowspan=2, columnspan=2,
                sticky = Tkinter.N+Tkinter.S+Tkinter.E+Tkinter.W)
```

The `N`, `S`, `E`, and `W` are acronyms for north, south, east, and west (in that order). In a cell, that's the top, bottom, right, and left sides.

This example's widgets stick to all four sides. You can choose which side(s) to stick to by adding those sides together with `+` like you see in the sample code. `SE`, `SW`, `NE`, `NW`, and `EW` also work.



```

C:\Python27\python.exe
Type "help", "copyright", "credits" or "license" for more information.
>>> import Tkinter
>>> root_window = Tkinter.Tk()
>>> label1 = Tkinter.Label(root_window, text="Label 1")
>>> label2 = Tkinter.Label(root_window, text="Label 2")
>>> label3 = Tkinter.Label(root_window, text="Label 3")
>>> label4 = Tkinter.Label(root_window, text="Label 4")
>>> label1.grid(row=0, column=0) # every counter starts
>>> label2.grid(row=0, column=1)
>>> label3.grid(row=1, column=0, columnspan = 2)
>>> label4.grid(row=0, column=2, rowspan=2, columnspan=2)
>>> label1.config(background="green")
>>> label2.config(background="red")
>>> label3.config(background="white", background="black")
>>> tk_rgb_template = "#%02x;%02x;%02x"
>>> red = 0
>>> green=128
>>> blue=128
>>> tkinter_teal = tk_rgb_template%(red, green, blue)
>>> label4.config(background=tkinter_teal)
>>> label3.grid(row=1, column=0, columnspan = 2, sticky = Tkinter.N+Tkinter.S+Tkinter.E+Tkinter.W)
>>> label4.grid(row=0, column=2, rowspan=2, columnspan=2, sticky = Tkinter.N+Tkinter.S+Tkinter.E+Tkinter.W)
    
```

Figure 11-13: Use `sticky` to expand the widgets.

Stretch with the pack geometry

When you use a pack geometry, you've got to solve the problem a different way:

- ✓ By telling Tkinter whether the widget expands when the window it's in is resized.
- ✓ By telling Tkinter whether the widget fills the available space (if there is any).

1. Type this code in your Python (command line) instance to set up the environment:

```

>>> import Tkinter
>>> root_window_pack = Tkinter.Tk()
>>> label1 = Tkinter.Label(root_window_pack, text="Label 1")
>>> label1.pack()
>>> label1.config(background="green")
    
```

By now the area around the label should be green.

2. Type the following code to move and resize the window:

```

>>> new_geometry_template= "%dx%d%d%d"
>>> new_geometry = new_geometry_template%(100,100,100,100)
>>> root_window_pack.geometry(new_geometry)
    
```

3. Pack the `label` widget again, passing the option `expand=1` (`expand=True` also works).

The widget moves to the center of the window. When a parent widget has more space than it needs to hold its child widgets, that extra space is shared out to those child widgets that were packed with `expand=1`. The parent widget (here, `root_window_pack`) has been resized so it's bigger than necessary. When the label is repacked, that extra space is given to the label. Now it fills the parent widget. That's why it's centered.

```
>>> label1.pack(expand=1)
```

4. Pack the label with `fill=Tkinter.X`.

This fills the background color horizontally across the space assigned to the widget (as opposed to the area where the writing is). Packing with `Tkinter.Y` makes it fill vertically, and `Tkinter.BOTH` causes the fill both horizontally and vertically. Each of these (`Tkinter.X`, `Tkinter.Y` and `Tkinter.BOTH`) is a constant in the `Tkinter` module:

```
>>> label1.pack(fill=Tkinter.X)
>>> label1.pack(fill=Tkinter.Y)
>>> label1.pack(fill=Tkinter.BOTH)
```

See Figure 11-14 for `label1.pack(fill=Tkinter.X)`. See Figure 11-15 for `label1.pack(fill=Tkinter.BOTH)`.

It gets a little trickier when `Tkinter.Frames` are involved. If you want a widget to expand, you have to make sure its parent widget, such as a `Tkinter.Frame`, also expands. (This applies to all parent widgets between the child widget and the root window.)

In this example, you look at similar `expand` and `fill` options, but this time the widget (`label1`) is a child of another widget.

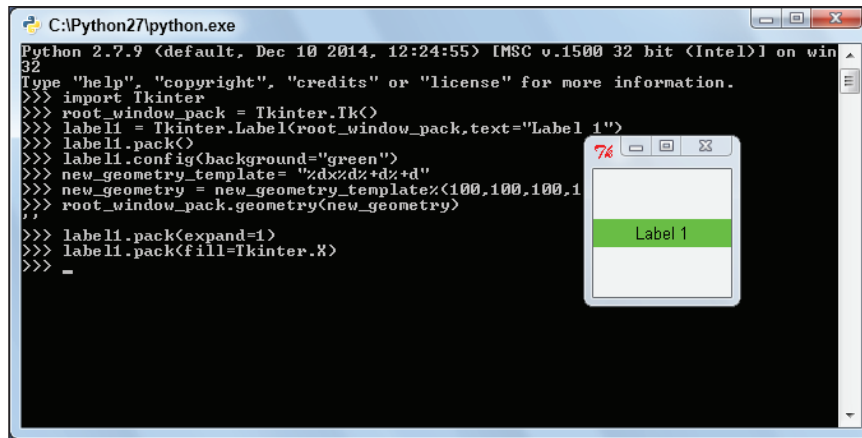


Figure 11-4: The label fills the available space horizontally.

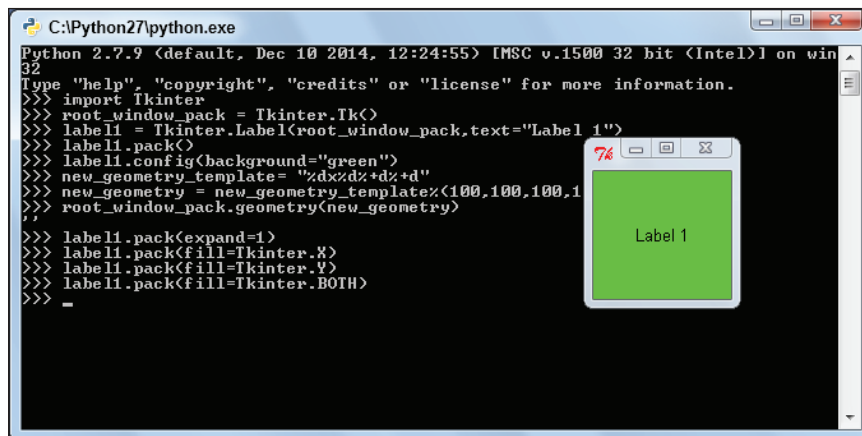


Figure 11-15: The label fills the available space both vertically and horizontally.

1. Set up the environment:

```

>>> import Tkinter
>>> root_window_pack = Tkinter.Tk()
>>> frame = Tkinter.Frame(root_window_pack)
>>> label1 = Tkinter.Label(frame, text="Label 1")
    
```

The label is added as a child of the widget `frame`, not `root_window_pack`.

2. Make the label's background green, pack the frame widget and the label widget, and resize the root window:

```
>>> label1.config(background="green")
>>> frame.pack()
>>> label1.pack()
>>> new_geometry = "%dx%d%d%d"%(100,100,100,100)
>>> root_window_pack.geometry(new_geometry)
```

See Figures 11-16 and 11-17.

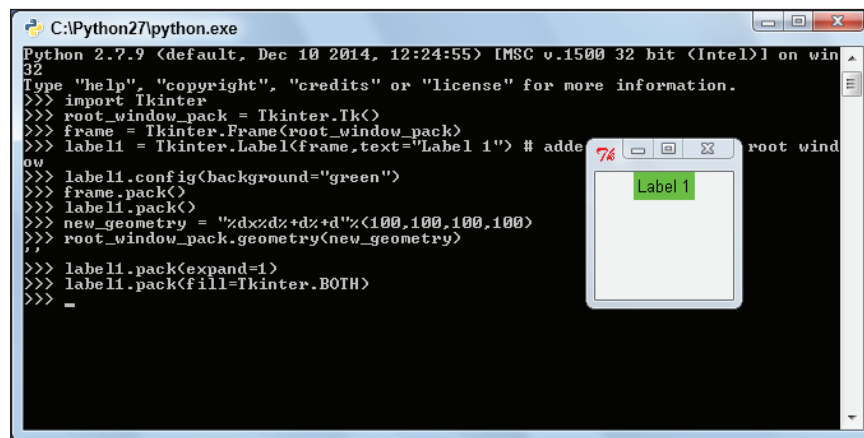


Figure 11-16: The label's in the same spot and it's linked to the frame.

If you set it, that extra space is allocated to frame and to label1.

```
>>> frame.pack(expand=1)
>>> frame.pack(fill=Tkinter.BOTH)
```



Extend Tkinter with ttk

Tkinter has an extension called `ttk`. The `ttk` module is part of the standard library. It includes newer versions of the Tkinter widgets and adds some others, including a tree view widget. To find out more, import `ttk` and use your Python introspection skills!

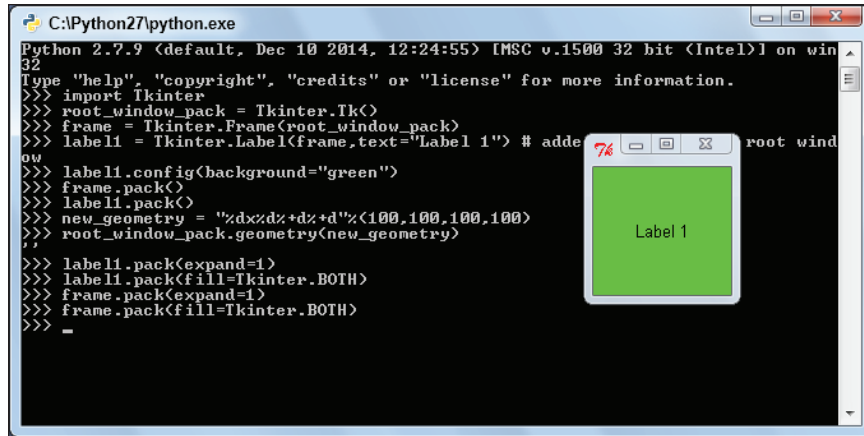


Figure 11-17: When the frame expands, the label expands as well.

Use Dialog Boxes

A *dialog box* is a small window that lets users know something. You tend to use dialog boxes (sometimes called plain old *dialogs*) when the application can't continue without information from the user.

Message boxes that stop the main application are called *blocking* or *modal* dialog boxes. Tkinter comes with some standard dialog boxes — `tkMessageBox` (I know it's called a message box, but it's a dialog box) and `tkFileDialog`. Each module has methods to display dialog box variants.

`tkMessageBox`

This module has a variety of methods, and they all format a title string and message string differently. The methods follow:

```

showinfo
showwarning
showerror
askquestion
    
```

```
askokcancel
askretrycancel
askyesno
askyesnocancel
```

Running one of these methods displays a dialog box. The user needs to click one of the buttons for the application to keep going.

- ✓ The show methods only give an OK button. The return value from these is always "ok", so don't bother capturing it.
- ✓ The ask methods provide different buttons depending on the method. Return values from the ask methods are "yes", "no", True, False, or None.

Here's some sample code for each of these message boxes. Create a new file (call it `testing_dialogs.py`) and put this code in it. When it runs, you'll see each message box option, one after another.

If you want, you can type these directly into your Python (command line) prompt. Each one gives you a different combination of icons and buttons. Some of them have return values of True or False, while others return 'yes', 'no', or 'ok'. These dialog boxes are all modal.

```
import Tkinter
import tkMessageBox
parent_window =Tkinter.Tk()

result = tkMessageBox.showinfo("Title","A Message")
print(result)
result = tkMessageBox.showwarning("Title","A Warning")
print(result)
result = tkMessageBox.showerror("Title","An Error")
print(result)
result = tkMessageBox.askquestion("Title", "A Question")
print(result)
```

```
result = tkMessageBox.askokcancel("Title", "A Question")
print(result)
result = tkMessageBox.askretrycancel("Title", "A Question")
print(result)
result = tkMessageBox.askyesno("Title", "A Question")
print(result)
result = tkMessageBox.askyesnocancel("Title", "A Question")
print(result)
parent_window.destroy()
```

tkFileDialog

This module brings up a dialog box that lets users choose a file to open (`askopenfilename`), to save (`asksaveasfilename`), or a number of other things. By passing options to the dialog box, you can specify a directory to start in, a file to select or suggest, and a list of file types to display.



Save the return value you get when you call these widgets. It gives you a path to the file that the user chose (or an empty string if you cancel). The `asksaveasfilename` method warns you before letting you choose an existing file, so that you don't accidentally delete it.

These dialog boxes ask the user for the name of a file to open (for `askopenfilename`) or to save to (for `asksaveasfilename`). Save the return value so you can use it to open the right file or save to the right filename. The `tkFileDialog` just returns a file's name and path. It doesn't do the saving for you. You have to do that yourself.

This code shows each of these dialog boxes in action. When it pops up, select a file before clicking OK. (It's okay. This code doesn't change those files.) Look at the result that prints to understand the format Python is returning the relevant values.

```
import Tkinter
import tkFileDialog
parent_window =Tkinter.Tk()
```

```
result = tkFileDialog.askopenfilename()
print result
result = tkFileDialog.asksaveasfilename()
print result

parent_window.destroy()
```



A piece of code that coders aren't supposed to use anymore is *deprecated*. Code is usually deprecated because someone has come up with a better way of doing things. You can't just dump the code because applications may depend upon it. It's deprecated so people remove it over time.

Summary

This project is a crash course in Python GUI programming. You

- ✓ Created your first Hello GUI World! application using a label widget.
- ✓ Met widgets and know that you must pack them before they're visible (when using the pack geometry manager).
- ✓ Discovered the online documentation for Tkinter — and found out how to pronounce it!
- ✓ Used the Python command line to program Tkinter interactively.
- ✓ Read about the root (also known as *parent* or *top*) level window and how to create it with the Tk() function.
- ✓ Read about Tkinter event management with the mainloop function.
- ✓ Registered callbacks and bound an event to a callback.
- ✓ Saw different event categories.

Bonus Project 1: Hello GUI World! **BC39**

- ✓ Exited a Tkinter application by using the `destroy` method on the root window.
- ✓ Discovered two ways to locate widgets in an application — with the `pack` geometry manager or the `grid` geometry manager.
- ✓ Knew *not* to use both `grid` and `pack` managers in the same top-level window.
- ✓ Colored and stretched your widgets.
- ✓ Used the standard dialog boxes in `tkMessageBox` and `tkFileDialog`.

