Unit 8

Learn something new

1. In Program 6 of Unit 4, we collected functions for working with playing cards into a module called **cards.py**. Here's part of what this module might contain. In what follows, we'll assume that **cards.py** includes a **shuffledDeck** function like this one that returns a list of 52 strings in random order, each representing one card in a shuffled deck.

```
import random

faceValues = ['ace', '2', '3', '4', '5', '6',
          '7', '8', '9', '10', 'jack',
          'queen', 'king']

suits = ['clubs', 'diamonds', 'hearts',
       'spades']

def shuffledDeck():
   deck = []
   for faceValue in faceValues:
      for suit in suits:
         deck.append(faceValue + ' of ' + suit)
   random.shuffle(deck)
   return deck
```

   Now consider the following program, which takes a very new approach to choosing a random card. If possible, check that **cards.py** contains a correct version of **shuffledDeck** and then run this new program before reading any further.

```
from tkinter import *
import cards

def firstCard():
   deck = cards.shuffledDeck()
   card['text'] = deck[0]

root = Tk()

card = Label(root)
card.pack()

pick = Button(root)
pick['text'] = 'Pick a card'
pick['command'] = firstCard
pick.pack()
```

```
mainloop()
```

There's a great deal that's new in this program, but what's most striking is what happens when it runs. Instead of using the IDLE shell window, this program has its own window and, inside, a button to click to control it. We say that it has a **graphical user interface** (GUI, say *gooey*), just like most familiar commercial programs.

To construct a GUI in Python, we typically use the `tkinter` module. We could write

```
import tkinter
```

as usual, but then we would have to prefix all calls to functions in the module with `tkinter`, like this:

```
tkinter.Tk()
tkinter.Label()
tkinter.Button()
```

Since we're going to be using many such functions, this would get to be tedious. Instead, we write

```
from tkinter import *
```

This makes all names from the module—including function names—available directly, no prefix necessary.

Our GUI programs will always start with a call to the `Tk` function. This creates the main window for the running program. We'll assign a name to this window—normally `root`—so that we can refer to it. Then we'll create **widgets** and add them to the main window. A widget is one element of a GUI: a button that can be clicked to make something happen, a box in which the user can enter text, an image, a menu, etc.

In our example, there are two widgets. The first is a label that we call `card`. A label is just a rectangular area where one line of information can be displayed. The label called `card` is where the name of a chosen card will be shown. Note that when we create the label, by calling `tkinter`'s function `Label`, we pass in `root`. Every widget will be part of some existing element of the interface. In this case, passing `root` to `Label` specifies that the new label will be part of the main window.

Another point to note carefully is that creating a widget does not, immediately, add it to the GUI under construction. To do that, we call its `pack` method. Soon we'll see how we can specify how `pack` decides where and how to place the widget, so that we

can control the layout of our GUI. For now, though, the key point is that a widget must be both created and packed before it will be visible.

The second widget is a button called **pick**. The first of the four lines at the bottom of the program creates the button. The last packs it.

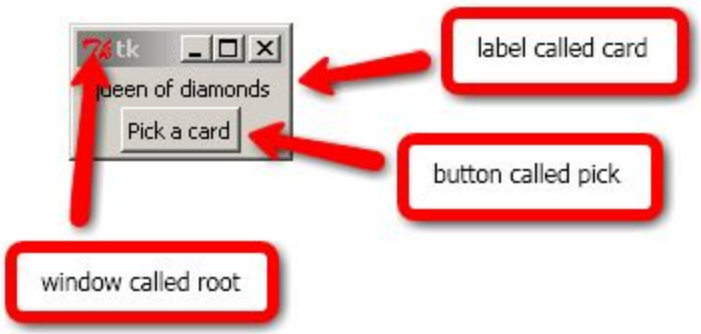
The middle two lines associate values with properties of the button, just as if it were a dictionary and the properties were keys. The **text** property of a button is what is displayed on it—in this case 'Pick a card'. Naturally, we label buttons with text that indicates what will happen when they're clicked.

The **command** property of a button specifies a function to be called when the button is clicked. In our case, clicking the button will cause a call to the **firstCard** function. Note carefully that we did *not* write

```
pick['command'] = firstCard()
```

This would set the **command** property to the result of *calling* **firstCard**, that is to something like the string '**2 of spades**'. Note also that the value of the **command** property must be a function that does not take arguments. We'll discuss later how to work around this rule.

When the user does click the button in our program, **firstCard** creates a shuffled deck and sets the **text** property of **card** to the string representing the first card in the deck. The **text** property of a label is what's displayed in it, so this causes the chosen card to appear in the window. Here's how the running program looks:



2. After finishing the work of displaying a window with a button in it, our card program is not done. In fact, it is just beginning its useful existence. The window remains on the screen responding to user clicks until the user chooses to close it. But then what instructions is the program following in order to control the window's behavior?

   The answer is that in a GUI program based on **tkinter** we normally include a call at the end to the **mainloop** function, which takes over control after the other instructions

we have written are carried out. The `mainloop` function starts an **event loop**. After all the widgets are in place, what we want is for our program to wait for something relevant to happen. The event loop looks conceptually like this:

```
while                    :
  if              :
```

That is, check over and over if something happened and, if so, respond appropriately.

Even in the case of our simple program, there are many relevant events. The obvious one is if the user clicks the `pick` button. But our program also responds correctly if the user minimizes the associated window or resizes it or closes it. Each of these is an event. We specified what the program should do in the case of a button click. All of the other event responses are provided by default—subject to change if the defaults don't satisfy us.

By the way IDLE is itself a GUI program. Up till now, we've always run our programs from within IDLE. But now that isn't necessary. To run the pick-a-card program directly, just find the program file on your computer and double-click it.[1]

3. As a second GUI example, we're going to write a trading game. The player starts with $10,000 in cash and watches as the price of a certain stock goes up and down. At any time, the player may click on a 'Buy' button to purchase 10 shares of the stock at the current price or click on a 'Sell' button to sell 10 shares purchased earlier. By waiting to buy until the price is low and then waiting for prices to rise before selling, the player hopes to make money.

   We'll be dividing the window into two halves, one containing information about the player's status and another containing the price and the 'Buy' and 'Sell' buttons. Each half will consist of a frame widget, which is simply a rectangle into which other widgets may be placed. Frames help us with layout and organization.

   Let's start with just the `status` frame and see how we can control the way it's packed. Here's code to create the frame and to place a label inside it.

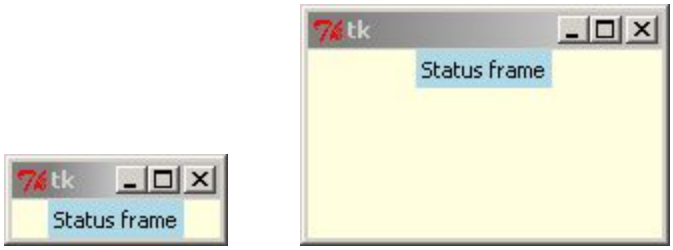
```
from tkinter import *

root = Tk()
```

[1] On Windows, it's also a good idea to change the extension from `.py` to `.pyw`. Then the program runs without opening a shell window in addition to the GUI. Note that double-clicking the program won't work unless Python is installed. That is, you can't give the file as is to a friend and have him or her run it on a different machine, unless that machine already has Python on it. For this you need a frozen binary. For Windows, see `py2exe.org`. Note, however, that as of July 2009, this is not yet available for Python 3.

```
root['bg'] = 'light yellow'

status = Frame(root)
status['bg'] = 'light green'

ID1 = Label(status)
ID1['text'] = 'Status frame'
ID1['bg'] = 'light blue'
ID1.pack()

status.pack()

mainloop()
```

Note that we write `ID1 = Label(status)` rather than `ID1 = Label(root)`. We're placing the label inside the frame. Also, we've set the background color of the window, frame and label using the `bg` property. This is a temporary measure just to help make it clear which is which when the program is run.

Here's the result. The window is shown on the left as it first appears and on the right as it looks after we drag the corner to make it a little larger.

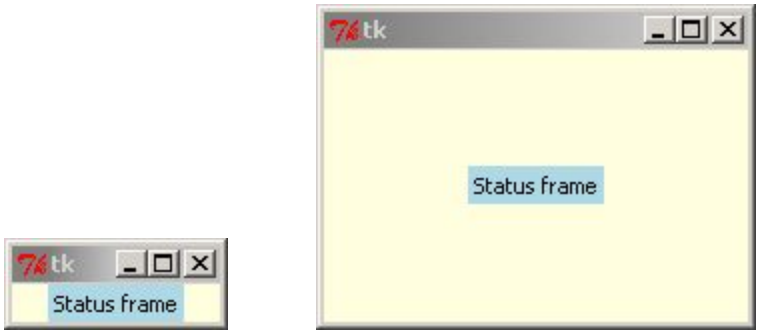


We can't see the green of the frame in either picture, because the blue label entirely fills it. Also, the picture on the right illustrates the fact that widgets are placed by default at the *top* of the **parent**—the object of which it is a part. The frame filled by the blue label is at the top of its parent, the main window.

Another point illustrated by the right-hand picture is that, by default, the space allocated to a widget does not change when its parent is resized. On the right, the main window is larger, but the frame and label still get the same space they did when it was small. We can change this, by modifying the last line of our code as follows:

```
status.pack(expand=YES)
```

This says that the status frame should be allocated as much space as possible. In our case—since there are no other widgets in the main window yet—the status frame gets *all* the space. The result is a little surprising though.

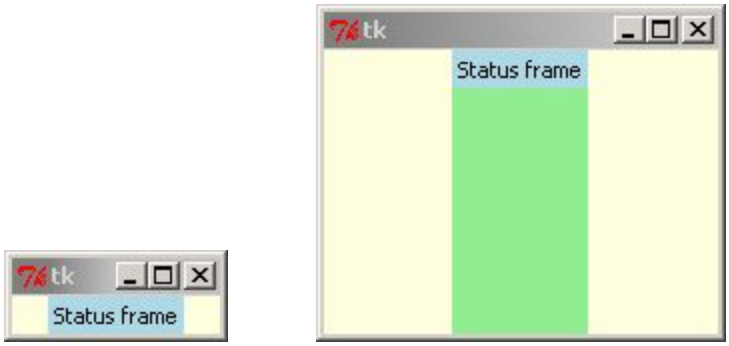


When the window is enlarged, the status frame is allocated all the space in the main window—that is, the space is *set aside* for its use. But the frame itself is no larger than before, so it ends up *using* only a small part of the space that's now available. By default, a widget goes in the center of the space allocated to it. What we're seeing is like a small car parked in the middle of large garage. The garage may be exclusively for the use of that particular car, but the car doesn't fill it.

If we want the status frame to fill the allocated space, we use the `fill` keyword argument. One possibility is to give it the value `Y`, as follows:

```
status.pack(expand=YES, fill=Y)
```

This says that the status frame should stretch vertically to fill the allocated space. Here's the result.



The entire window is available for the use of the green status frame. On the right, the frame has expanded vertically as far as its space permits. The blue label has not changed in size, since it is packed with default values. Also, we can now see for the first time that the label is packed, by default, at the top of the frame.

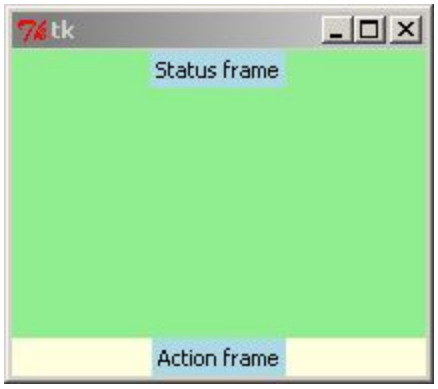
In addition to the value `Y` for vertical stretching, the `fill` keyword argument can be set to `X` for horizontal stretching or, most commonly, to `BOTH` so that it will stretch vertically and horizontally, filling all allocated space.[2]

Let's arrange for the status frame to stretch in both directions and also add our second frame, along with a label to identify it. Recall that this second frame will contain the stock price and the action buttons 'Buy' and 'Sell'. We'll call it the action frame. Here's our modified code:

```
from tkinter import *

root = Tk()
root['bg'] = 'light yellow'

status = Frame(root)
status['bg'] = 'light green'

ID1 = Label(status)
ID1['text'] = 'Status frame'
ID1['bg'] = 'light blue'
ID1.pack()

action = Frame(root)
action['bg'] = 'pink'

ID2 = Label(action)
ID2['text'] = 'Action frame'
ID2['bg'] = 'light blue'
ID2.pack()

status.pack(expand=YES, fill=BOTH)
action.pack()

mainloop()
```

And here's the result, after the window is enlarged:

[2] These names and others in capital letters are imported from `tkinter`. For example, the `tkinter` module includes the following line of code: `X = 'x'`. Since we've imported all names from the module, we can write `X` instead of `tkinter.X`.

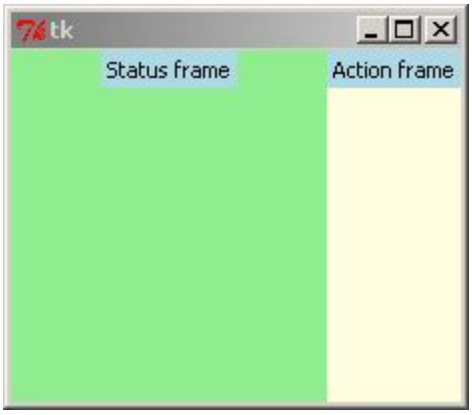


Notice that the space allocated to the status frame expands as far as possible, leaving the action frame only the bare minimum necessary, and also that the status frame stretches to actually fill all the space it owns.

Another crucial keyword argument to `pack` is `side`, which allows us to control the placement of a widget. The default value is `TOP`, but we can also use `LEFT`, `RIGHT` or `BOTTOM`. Let's put the status frame at the left, by modifying the code as follows:

```
status.pack(side=LEFT, expand=YES, fill=BOTH)
```

Here's the result:

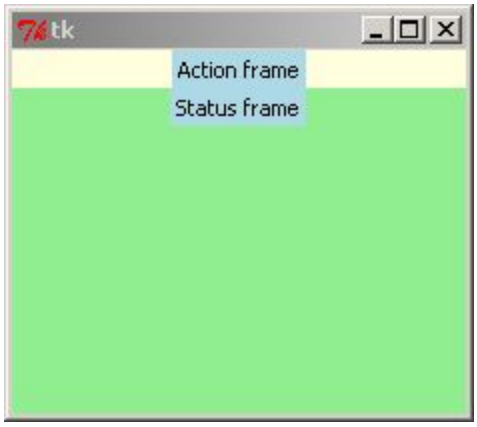


Note carefully that the action frame is placed at the top, by default, but *not* at the top of the whole window. Instead, the status frame is packed first, at the left and then the status frame is packed at the top of the *remaining* space.

If we simply reverse the order of the packing statements, like this

```
action.pack()
status.pack(side=LEFT, expand=YES, fill=BOTH)
```

we get a very different result:



In this case, the action frame is packed first, at the top of the main window, and the status frame is packed at the left of what remains. Of course, since the status frame is set to expand its domain and fill it as far as possible, it reaches from the left side where it is packed all the way across to the right.

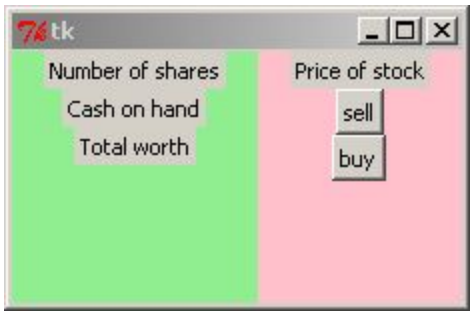
4. At this point, we're ready to take control of the layout of our trading game. We'll place the status frame on the left and the action frame on the right, letting both expand as far as possible. Since they're competing for space, each will get about half. We'll add labels in the status frame that will soon be used to display the number of shares the player owns, the amount of cash he or she has and the total value of the cash and stock combined. In the action frame we'll add a label to display the current stock price and the 'Buy' and 'Sell' buttons.

   Here's the code.

```
from tkinter import *

root = Tk()
root['bg'] = 'light yellow'

status = Frame(root)
status['bg'] = 'light green'

shares = Label(status)
shares['text'] = 'Number of shares'
shares.pack()

cash = Label(status)
cash['text'] = 'Cash on hand'
cash.pack()

worth = Label(status)
worth['text'] = 'Total worth'
worth.pack()
```

```
action = Frame(root)
action['bg'] = 'pink'

price = Label(action)
price['text'] = 'Price of stock'
price.pack()

sell = Button(action)
sell['text'] = 'sell'
sell.pack()

buy = Button(action)
buy['text'] = 'buy'
buy.pack()


status.pack(side=LEFT, expand=YES, fill=BOTH)
action.pack(side=RIGHT, expand=YES, fill=BOTH)

mainloop()
```

And here's the result:

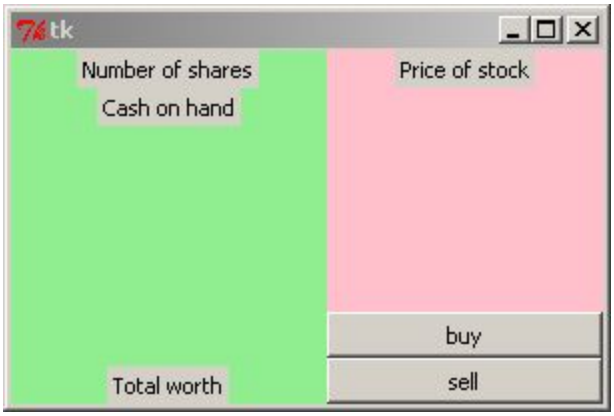


We can set off the label for the total worth and the action buttons by moving them to the bottom. Also, let's allow the buttons expand horizontally so that they end up with the same width, the full width of the action frame. Here's the modified code

```
from tkinter import *

root = Tk()
root['bg'] = 'light yellow'

status = Frame(root)
status['bg'] = 'light green'

shares = Label(status)
shares['text'] = 'Number of shares'
shares.pack()
```

```
cash = Label(status)
cash['text'] = 'Cash on hand'
cash.pack()

worth = Label(status)
worth['text'] = 'Total worth'
worth.pack(side=BOTTOM)


action = Frame(root)
action['bg'] = 'pink'

price = Label(action)
price['text'] = 'Price of stock'
price.pack()

sell = Button(action)
sell['text'] = 'sell'
sell.pack(side=BOTTOM, fill=X)

buy = Button(action)
buy['text'] = 'buy'
buy.pack(side=BOTTOM, fill=X)


status.pack(side=LEFT, expand=YES, fill=BOTH)
action.pack(side=RIGHT, expand=YES, fill=BOTH)

mainloop()
```

And here's the result.



Finally, notice that each label is centered within its allocated space. We can change this—for any widget, not just a label—by changing the default value of the **anchor** keyword in the call to **pack**. Other than the default, **CENTER**, the options correspond to the eight points on a compass: **N**, **NE**, **NW**, **W**, **E**, **SW**, **S** and **SE**. We'll move the

labels in the status frame to the west (left) side and the price label to the east (right). Here's the code. Note that we've removed the coloring, since its only purpose was to help us understand the layout.

```
from tkinter import *

root = Tk()

status = Frame(root)

shares = Label(status)
shares['text'] = 'Number of shares'
shares.pack(anchor=W)

cash = Label(status)
cash['text'] = 'Cash on hand'
cash.pack(anchor=W)

worth = Label(status)
worth['text'] = 'Total worth'
worth.pack(side=BOTTOM, anchor=W)

action = Frame(root)

price = Label(action)
price['text'] = 'Price of stock'
price.pack(anchor=E)

sell = Button(action)
sell['text'] = 'sell'
sell.pack(side=BOTTOM, fill=X)

buy = Button(action)
buy['text'] = 'buy'
buy.pack(side=BOTTOM, fill=X)


status.pack(side=LEFT, expand=YES, fill=BOTH)
action.pack(side=RIGHT, expand=YES, fill=BOTH)

mainloop()
```

And here's the result, our final layout:

5. Now that the program is visually correct, let's make it act as it's supposed to. First, we'll add names for all the values we need to keep track of and a function called **update** that displays each value, neatly formatted, in the appropriate label.

```
from tkinter import *

numberShares = 0
account = 10000
sharePrice = 97

def update():
    shares['text'] = 'You own {0} shares'.format(numberShares)
    cash['text'] = 'Cash balance: ${0:.0f}'.format(account)
    totalWorth = account + numberShares*sharePrice
    worth['text'] = 'Total worth: ${0:.0f}'.format(totalWorth)
    price['text'] = '${0:.2f}/share'.format(sharePrice)

root = Tk()
    .
    .
    .
status = Frame(root)
status.pack(side=LEFT, expand=YES, fill=BOTH)
action.pack(side=RIGHT, expand=YES, fill=BOTH)

update()

mainloop()
```

Here's the result:

Next, we add functions that buy and sell 10 shares of stock and associate these with the **command** properties of the 'Buy' and 'Sell' buttons.

```
from tkinter import *

numberShares = 0
account = 10000
sharePrice = 97

def update():
   shares['text'] = 'You own {0} shares'.format(numberShares)
   cash['text'] = 'Cash balance: ${0:.0f}'.format(account)
   totalWorth = account + numberShares*sharePrice
   worth['text'] = 'Total worth: ${0:.0f}'.format(totalWorth)
   price['text'] = '${0:.2f}/share'.format(sharePrice)

def doBuy():
   global account, numberShares
   if account >= 10*sharePrice:
      numberShares += 10
      account -= 10*sharePrice
      update()

def doSell():
   global account, numberShares
   if numberShares >= 10:
      numberShares -= 10
      account += 10*sharePrice
      update()

root = Tk()

status = Frame(root)
      .
      .
      .
sell = Button(action)
sell['text'] = 'sell'
sell['command'] = doSell
sell.pack(side=BOTTOM, fill=X)
```

```
buy = Button(action)
buy['text'] = 'buy'
buy['command'] = doBuy
buy.pack(side=BOTTOM, fill=X)


status.pack(side=LEFT, expand=YES, fill=BOTH)
action.pack(side=RIGHT, expand=YES, fill=BOTH)

mainloop()
```

In the **doBuy** function, we check to make sure the player has enough cash on hand to buy ten shares at the current price before proceeding with the transaction. This way, if the 'Buy' button is clicked when the player doesn't have enough money to make the purchase, nothing will happen. Likewise, in **doSell**, we check to make sure the player actually owns at least 10 shares before allowing the desired sale to proceed—you can't sell if you don't have any shares.

After either a purchase or sale, we call the **update** function, so that the display will reflect the latest information. Both of the new functions also include a **global** statement. This is necessary, but we'll postpone discussion of why till a little later.

Here's how the program looks after we've clicked 'Buy' three times.

Of course, since the price hasn't changed, the 30 shares purchased for $97 each are still worth $97. The player has $7,090 cash in hand, plus 30 shares worth $97 each, for a total worth of $10,000. Owning shares can't affect the player's worth until the price begins to change.

6. We'd like to have the price change on a regular basis, say every two seconds. The **tkinter** module includes a function called **after** that can be used for this kind of scheduling. We can call it as a method of either the main window, like this

```
root.after(2000, fn)
```

or of any widget, like this

```
cash.after(2000, fn)
```

In either case, the effect is to cause the function named as the second argument to be called after a delay measured in milliseconds and specified by the first argument. In our example, `fn` will be called once after 2 seconds.[3]

To get an unending series of calls to a function, we use a standard trick. The last line of code in the called function schedules the next call to the same function. Here's how the code might look:

```
def fn():

  root.after(2000, fn)
```

Now if we call `fn` once directly, a new call will automatically be scheduled for two seconds later. When this second call finishes, it will schedule a third call. Since this process never ends, we'll get a steady stream of calls, once every two seconds.

Here's the final code for our game:

```
import random
from tkinter import *

numberShares = 0
account = 10000
sharePrice = 97

def update():
   shares['text'] = 'You own {0} shares'.format(numberShares)
   cash['text'] = 'Cash balance: ${0:.0f}'.format(account)
   totalWorth = account + numberShares*sharePrice
   worth['text'] = 'Total worth: ${0:.0f}'.format(totalWorth)
   price['text'] = '${0:.2f}/share'.format(sharePrice)

def doBuy():
   global account, numberShares
   if account >= 10*sharePrice:
      numberShares += 10
      account -= 10*sharePrice
      update()

def doSell():
   global account, numberShares
   if numberShares >= 10:
      numberShares -= 10
      account += 10*sharePrice
```

[3] A millisecond is 1/1000th of a second. So 1000 milliseconds is one second, 5,000 milliseconds is five seconds, 500 milliseconds is half a second and so on.

```
    update()

def changePrice():
    global sharePrice
    sharePrice += random.random()*4 - 2
    update()
    root.after(2000, changePrice)

root = Tk()

status = Frame(root)

shares = Label(status)
shares.pack(anchor=W)

cash = Label(status)
cash.pack(anchor=W)

worth = Label(status)
worth.pack(side=BOTTOM, anchor=W)

action = Frame(root)

price = Label(action)
price.pack(anchor=E)

sell = Button(action)
sell['text'] = 'sell'
sell['command'] = doSell
sell.pack(side=BOTTOM, fill=X)

buy = Button(action)
buy['text'] = 'buy'
buy['command'] = doBuy
buy.pack(side=BOTTOM, fill=X)

status.pack(side=LEFT, expand=YES, fill=BOTH)
action.pack(side=RIGHT, expand=YES, fill=BOTH)

changePrice()

mainloop()
```

A few small items are worth pointing out:

- The `changePrice` function includes a call to `update` so that the display will change immediately to reflect the new price.

- We've eliminated code setting initial text values for the labels, since the call to `changePrice` will immediately replace them.

- `changePrice` function calls the function `random` from the `random` module, that is `random.random`. To get access to the function, we import the module.

We also need to take a moment to see just *how* the stock price is being changed. The `random` function returns a random number between 0 and 1. If we multiply the result by 4, we get a result from 0 to 4. If we then subtract 2, we get a result from -2 to 2. Adding this to `sharePrice` thus makes it go up or down randomly by up to two dollars.

By the way, you might wonder who is responsible for making sure that scheduled calls to `changePrice` are actually carried out? It's the event loop. Scheduled calls are another example of events that occur and to which our program responds.

7. In many of our program so far, we liberally reuse names. Here's an example:

```
def nextYear(year):
   year += 1
   print('***', year, '***')

def decade(start):
   for year in range(start, start+10):
      nextYear(year)

year = 1832
nextYear(year)
print()
decade(1492)
print()
print(year)
```

Here we've used the name `year` in the main part of the program to refer to the integer 1832. If you run the program, you'll see that `year` still has this meaning in the last line—it displays 1832. Of course this is no surprise, but how can we account for the fact that this is true even though `year` is reassigned to refer to the integer 1833 in the call to `nextYear` and that it is assigned to ten other integers in the call to `decade`?

The answer is that this program actually uses three completely separate names. One is the name `year` used in the main program to refer to 1832. Another is the name `year` used only within the `nextYear` function. It takes on a value whenever a call is made to `nextYear` and, within the function, it is reassigned to the next higher number. The third name is also `year` and it exists only within the `decade` function. Within that function, it is assigned to a series of integers by the `for` statement.

Several people may share the name John or Emily, although they have nothing else to do with one another. Certainly if one John or Emily has a birthday, the others are

unaffected. Likewise, what we do with names that are **local** to a function—defined within it—has no effect on an identical name in another function. It also has no effect on a **global** name—one defined outside any function.

Occasionally, however, we *do* want a function to make a change in a global name. In this case, we must identify the name as global using a `global` statement. Here's an example:

```
def nextYear(year):
   year += 1
   print('***', year, '***')

def makeItBeNextYear():
   global year
   year += 1

year = 1832

nextYear(year)
print(year)

makeItBeNextYear()
print(year)
```

After the call to `nextYear`, the name `year` in the main program still refers to 1832, despite the reassignment within the function. After the call to `makeItBeNextYear`, however, the name `year` in the main program refers to 1833. The simple rule is: *If a global name is to be reassigned within a function it must be identified as global*.

This explains why we needed to use `global` statements in our trading game. The `doBuy`, `doSell` and `changePrice` functions all reassign global names.

Note that it is not necessary to identify a name as global if it will be used, but not reassigned. For example, the `update` function in our game uses many global names, without reassigning any:

```
def update():
   shares['text'] = 'You own {0} shares'.format(numberShares)
   cash['text'] = 'Cash balance: ${0:.0f}'.format(account)
   totalWorth = account + numberShares*sharePrice
   worth['text'] = 'Total worth: ${0:.0f}'.format(totalWorth)
   price['text'] = '${0:.2f}/share'.format(sharePrice)
```

The global names `account`, `numberShares` and `sharePrice` are clearly not reassigned within this function. But what about the global name `shares` and the others used to

refer to widgets? These aren't reassigned either. The name `shares` starts out referring to a certain label widget and it ends up referring to the same widget. Only a *property* of the widget has been changed.