Lesson 18: Animation

So how do you make your drawings move and change? That's what this section is about.

I'd like to introduce you to your new friend, Mr. Timer. A timer gives you the ability to tell your DrawingWindow to run some code again and again at regular time intervals. This is how you make things moved at a controlled speed.

To start a timer, use the onTimerCall() method on DrawingWindow. Here is a description of `onTimerCall()`:

`dw.onTimerCall(interval_sec, callback)`

Set a timer that will call a function at regular intervals.

`dw` is a `DrawingWindow` object.

`interval_sec` is the time interval of the timer, in seconds.

`callback` is the function or method that the timer will call.

Return a timer object. You can stop the timer by calling the `stop()` method on the timer object.

In general here are the steps for using a timer. (We will do a specific example in just a moment.)

1. Think about what code you want the timer to run for you. Decide on a name for a function to hold this code.
2. Decide how often you want the function to be called.
3. Decide what data the function needs to use. If the function needs to modify data, then it is better to use a method on a class instead of a function. (I'll show an example of using a class later, in bounceball.py.)
4. Write the function or method.
5. Call `dw.onTimerCall(interval_sec, callback)` where **`interval_sec`** is the how often you want the function called, and **`callback`** is the name of your function or method (without parenthesis).

Here is our first timer example:

1. I have decided that I want the timer to call code that will draw a blue box and print the words “drawing a box.” I will name the function that does this `drawbox()`.
2. I want the timer to call my `drawbox()` function every 5 seconds.
3. The data that my function needs is a DrawingWindow object, because you can't draw anything without a drawing window. I will use a global variable, **`dw`**, for DrawingWindow, as usual.
4. In the commands below we will define the `drawbox()` function.
5. In the commands below we will call `dw.onTimerCall()` with the correct parameters.

Try these commands in the Python Shell window to execute our first timer example.

```
>>> from cpif.graphics import DrawingWindow
>>> dw = DrawingWindow(100, 100)
>>> def drawbox():
          print “drawing a box”
          dw.box(10, 10, 90, 90, 'black', fill='blue')


>>> timer = dw.onTimerCall(5.0, drawbox)
```

A blank white DrawingWindow will appear. After a delay of about 5 seconds a blue square appears in the window, and the words “drawing a box” appear in the Python Shell window.

About 5 seconds after the blue square appears, Python prints “drawing a box” again. Every 5 seconds it will do that. To stop the timer from drawing the box over and over again, call the stop method on the timer, like this:

```
>>> timer.stop()
```

You may be interrupted by the “drawing a box message” while you are typing. Just keep typing, Python will still accept your command.

The Moving Ball Example

The **`moveball.py`** sample program moves a ball across the screen using a function called by a timer. The function name is `moveBall()`. The function needs two data items: The DrawingWindow object and the ID of a ball that the program drew.

Open the **`moveball.py`** program in the **`samples`** directory.

Here is the code:

```
# moveball.py
# Draw a ball and make it move


from cpif.graphics import DrawingWindow
dw = DrawingWindow(200, 200)
ball = dw.circle(0, 100, 20, 'black', fill='red')


def moveBall():
    dw.moveBy(ball, 10, 0) # move 10 pixels right


dw.onTimerCall(0.1, moveBall)
dw.run()


# end-of-file
```

Whoa, don't hit that F5 key yet! There are some other important things about this program I need to point out to you first.

Take a look at the `moveBall()` function. New to you is the `DrawingWindow` `moveBy()` method. Here is how to use it:

`dw.moveBy(id, deltaX, deltaY)`

`dw` is a `DrawingWindow` object.

`id` is an ID returned by a drawing method, such as `dw.circle()`.

`deltaX` and **`deltaY`** are the horizontal and vertical distance, in pixels, that the drawing will be moved.

In this example, **`deltaX`** is 10, meaning move 10 pixels right. If it were a negative number it would move the drawing left. **`deltaY`** is 0, meaning don't move it up or down at all.

Finally, please notice that the program ends with `dw.run()`. If you didn't call `dw.run()`, the program wouldn't wait while you looked at the window, but instead would exit immediately. You wouldn't see anything and it wouldn't be any fun. `dw.run()` pauses until the DrawingWindow is shut down, either by your program calling `dw.close()` or by you clicking the close button in the upper-right corner of the window frame.

Ok, that's enough explaining. Press F5 and let's see this thing roll.

Hey, where did that ball go? After about two seconds the red ball moved across the window, and then kept on going. It's completely out of view, but the timer keeps on running anyway.

A Thought Provoking Question

Here's a thought provoking question for you that's not part of the quiz: how would you modify this program to make the ball bounce off of the edges of the window? Right now the ball never changes speed nor direction. We need to find a way to store the speed and direction of the ball, detect when we hit the edge of the window, and change direction. How can we keep track of all that changing data?

The answer is, by creating our own Python class. I'll show how to do this in the **`bounceball.py`** example next.

The Bouncing Ball Example

Understanding this example requires a knowledge of Python classes. If you skipped the section on Classes , you might want to go back and read that now.

The Moving Ball Example in the previous section got boring very quickly, because after two seconds you are left with an empty window. One way we can fix that problem is to make the ball bounce back when it hits the edge of the window.

Of course the ball doesn't actually bounce, but we can simulate bounces by making it change direction. Let's talk a little more about speed and direction, before we get going on our bouncing ball example.

Normally we think of speed as a single number, that is, how fast you are going. But to simulate moving objects in a computer program we need to be a little more sophisticated. We'll use a more sophisticated word too: *velocity*. The velocity of an object moving in a two-dimensional plane must be measured by two numbers. We will use one number for velocity in the X (horizontal) direction, and another number for velocity in the Y (vertical direction.) In my examples I will often use the variables **`vx`** and **`vy`** as the names for these two numbers.

Numbers in the real world are seldom useful unless they are connected with units of measurement. Velocity is measured in units of distance per time, such as meters per second or miles per hour. In our example program, **`x`** and **`y`** are the location of the ball measured in pixels, and **`vx`** and **`vy`** are the velocity of the ball measured in pixels per update. The update time is determined by the interval in seconds that we specify when calling `dw.onTimerCall()`. In this program the update time in seconds is controlled by the **`delay`** variable.

When the ball goes off of the right or left edge of the window, the program will reverse the ball's horizontal velocity by changing the sign of **`vx`**, like this: `vx = -vx`. Likewise when the ball goes past the top or bottom edge of the window, the program will reverse the sign of **`vy`**. This will simulate bouncing. Our simulation will not be perfect, however, because the ball may go several pixels past the edge before the program detects that it has gone too far and reverses its direction. I know how to fix this problem, but doing so would make the program much more complicated. Let's just try it out the way it is.

Open the **`bounceball.py`** program in the **`samples`** directory.

Here is the code:

```
# bounceball.py
# Animate a ball, making it "bounce" off the edges of the window

from cpif.graphics import DrawingWindow

interval = 0.02 # interval between updates, in seconds
width, height = 200, 200 # size of window

class MovingBall:

    def __init__(self):
        # starting position of ball
        self.x = 100
        self.y = 100
        # starting velocity of ball
        self.vx = 1.4
        self.vy = -0.8
        # draw the ball and save its ID
        self.ball = dw.circle(self.x, self.y, 10, 'black', fill='red')

    def moveBall(self):
        dw.moveBy(self.ball, self.vx, self.vy)
        self.x = self.x + self.vx
        self.y = self.y + self.vy
        if self.x < 0 or self.x >= width:
            self.vx = -self.vx
        if self.y < 0 or self.y >= height:
            self.vy = -self.vy

dw = DrawingWindow(width, height)
mb = MovingBall()
dw.onTimerCall(interval, mb.moveBall)
dw.run()

# end-of-file
```

This program defines a class called MovingBall, and creates an instance of that class called **`mb`**. Notice the function that we passed to `dw.onTimerCall()` was a method on the MovingBall object, `mb.moveBall`. This is necessary in order to give the timer callback access to data that it can change. In this case it needs to change the position and the velocity of the ball, and remember that information from one update to the next.

The bouncing effect is accomplished by these lines in the `moveBall()` method:

```
        if self.x < 0 or self.x >= width:
            self.vx = -self.vx
        if self.y < 0 or self.y >= height:
            self.vy = -self.vy
```

Press F5 in the **`bounceball.py`** window and watch the little red ball bounce around for a while.

Mouse and Keyboard Input

This section shows how you can you can make a DrawingWindow respond to input from the mouse and the keyboard.

Making Buttons

You can create customized buttons that respond to mouse button clicks by following these steps:

1. Draw an object in a DrawingWindow, and save the object's ID.
2. Create a callback function or method that takes one parameter, **`event`**.
3. Call `dw.onClickCall()`, and pass the ID of the drawing, the callback function or method, and your data.

Type the following commands in the Python Shell to create a blue circle and turn it into a button that prints a green message:

```
>>> from cpif.graphics import DrawingWindow
>>> dw = DrawingWindow(100, 100)
>>> def showMessage(event):
```

```
          dw.drawText(0, 0, “Hello”, 'green')

>>> id = dw.circle(50, 50, 25, 'black', fill='blue')
>>> dw.onClickCall(id, showMessage)
```

When you run this code, a white window with a blue disk will appear. When you click the left mouse button on the blue disk you should see this result:

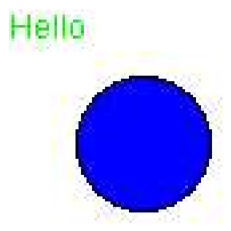

Illustration 10Result of clicking example button

There is some similarity between setting up a button and setting a timer. In both cases you create a function and then tell Python to call it later. Most graphical programming frameworks follow this general pattern; it's called *event-driven programming*.

There is a very important difference between `onClickCall()` and `onTimerCall()`: When you use `onClickCall()`, the first parameter that is passed to your callback function is always an event object. This event object is used to find the current mouse coordinates at the time you clicked the mouse button. This can be useful if you want to click and drag objects. When you write your own callback functions for handling clicks on drawings, follow the example of `showMessage()` above.

If you create a callback function with the wrong number of parameters (say, you forget the `event` parameter), then when you call `onClickCall()` or a similar method you will get an error that looks something like this:

```
Traceback (most recent call last):
...
ValueError: callback must be a function or method that takes 1
parameters
```

Check the **`callback`** function that you passed to `onClickCall()` (in this case, `callbackFunc()`) and make sure that it has an **`event`** parameter.

Here are the details about the DrawingWindow `onClickCall()` method:

`dw.onClickCall(id, callback)`

Set a function to be called when a drawing is clicked.

`dw` is a DrawingWindow object.

`id` is an ID number returned by a drawing method,
    or an ID returned by beginDrawing().
`callback:`
    The function to call when the left mouse button is pressed
    over the drawing. It will be called like this:
        `callback(mouse_event)`
    `mouse_event` is an object with the following attributes:
        `button_num`:
            1 for left button, 2 for middle, 3 for right,
            0 for no button
        `x`: the X window coordinate of the mouse event
        `y`: the Y window coordinate of the mouse event
    If **`callback`** is `None` then any function previously associated
    with this event is deactivated.

Responding to Key Presses

You can make your graphics program respond to keyboard commands by following these steps:

1. Write a callback function or method that takes one parameter, **`event`**. The function should look at **`event.char`** or **`event.keysym`** to find out which key has been pressed. (I will explain more about **`event.char`** and **`event.keysym`** later.)
2. Call the DrawingWindow `onKeyDownCall()` method and pass it your callback function or method, in order to activate your keyboard command.

Open the **`keynames.py`** program in your **`samples`** directory to see an example program that uses `onKeyDownCall()` and prints the **`event.char`** and **`event.keysym`** values every time you press a key. Here is the program:

```
# keynames.py
# Print the key character and symbol name as it is pressed.

from cpif.graphics import DrawingWindow

dw = DrawingWindow(480, 100)
dw.drawText(0, 0, "Click this window, then type on the keyboard.\n"
                  "See the key names in the text output window.")

def callback(event):
    print "Key pressed:",
    print " event.char==" + repr(event.char),
    print " event.keysym==" + repr(event.keysym)

dw.onKeyDownCall(callback)
dw.run()

# end-of-file
```

I pressed F5 to run the program, and then typed "test" and pressed Enter. I got the following result:

```
Key pressed:  event.char=='t'  event.keysym=='t'
Key pressed:  event.char=='e'  event.keysym=='e'
Key pressed:  event.char=='s'  event.keysym=='s'
Key pressed:  event.char=='t'  event.keysym=='t'
Key pressed:  event.char=='\r'  event.keysym=='Return'
```

When a key is a letter of the alphabet or a regular typewriter symbol, then **`event.char`** is the same as **`event.keysym`**, and is usually a string containing one character. When a key is a control character or something that makes the cursor move, etc. then **`event.keysym`** contains the name of that key (**`event.char`** will usually not be useful for control characters.) Notice that the **`event.keysym`** value for the Enter key is `'Return'` – that's what it is called on older typewriters.

Here are the **`event.keysym`** values for some of the control keys found on computer keyboards:

Key	***Value of event.keysym***
Esc	`'Escape'`
F1	`'F1'`
Left arrow	`'Left'`
Right arrow	`'Right'`
Up arrow	`'Up'`
Down arrow	`'Down'`

You can find the **`event.keysym`** values for the rest of the keys on your keyboard by running **`keynames.py`** and pressing the keys.

Example of Controlling a Program with the Keyboard

Some useful things you can do with keyboard commands are exit the program or move an object. The **`keymove.py`** program in your **`samples`** directory moves a little blue box in the window when you press the arrow keys, and exits the program when you press the Esc key. Here is the program:

```
# keymove.py
# Move the box using the arrow keys. Press Esc to exit.

from cpif.graphics import DrawingWindow

dw = DrawingWindow(640, 480)
dw.drawText(0, 0, "Use the arrow keys to move the box, Esc to quit")
box_id = dw.box(310, 230, 330, 250, 'black', fill='light blue')

def keyCallback(event):
    if event.keysym == 'Escape':
        dw.close()
    elif event.keysym == 'Left':
        dw.moveBy(box_id, -20, 0)
    elif event.keysym == 'Right':
        dw.moveBy(box_id, 20, 0)
    elif event.keysym == 'Up':
        dw.moveBy(box_id, 0, -20)
    elif event.keysym == 'Down':
        dw.moveBy(box_id, 0, 20)

dw.onKeyDownCall(keyCallback)
dw.run()

# end-of-file
```

Responding to Key Releases

There is a DrawingWindow method called `onKeyUpCall()` that works just like `onKeyDownCall()`, except that the callback function is called when the key is released instead of when the key is pressed. The Lander game example program at the end of the book uses this method.

FYI: "Appendix 3: Message Boxes" on page 162 contains more ways for displaying messages to the user and getting simple input from the user.

This lesson gives lots of examples. Hopefully you saw in the examples something similar to a program or part of a program that you'd like to write. Feel free to change and experiment with these sample program; that's a great way to learn.

Quiz 18

1. Adapt **`bounceball.py`** so that it makes the **`PythonPowered.gif`** image move and bounce around instead of the little red ball. Name it bouncelogo.py. (See “Drawing Images” on page 101.)
2. Write a program called **`digiclock.py`** that displays the current time in the upper-left corner of a window, updated once per second, using the DrawingWindow `onTimerCall()` method.

   Here is an example of how to get a string containing the current time:

   ```
   >>> import time
   >>> s = time.ctime()
   >>> print s
   Sat May 29 18:04:34 2004
   ```

   Here's a hint: each time the timer is called, you need to delete the previous text you have drawn before drawing the new time. Otherwise, your clock will look very strange. You'll need to use a class to keep track of the data, and have the timer call a method of that class, as in **`bounceball.py`**.
3. Write a program called **`clickme.py`** that draws a circle of radius 25 in the middle of a 200 by 200 window. When you click on the circle, it moves 25 pixels to the right. If you keep on clicking it, it moves completely out of sight.

   Reminder: The function or method that is called when you click the circle needs to take an `event` parameter as described above.