Eric Roberts
CSCI 121

Handout #11
September 14, 2018

Interactive Graphics

Interactive Graphics

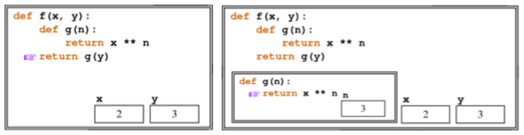
Eric Roberts
CSCI 121
September 14, 2018

Review: Nested Functions

- Python allows you define one function inside another, as in the following example from the text:

```
def f(x, y):
    def g(n):
        return x ** n
    return g(y)
```

- Calling `f(2, 3)` creates the following frames:



```
def f(x, y):
    def g(n):
        return x ** n
    return g(y)
```

```
def f(x, y):
    def g(n):
        return x ** n
    return g(y)
```

```
def g(n):
    return x ** n
```

Name Resolution and Scope

- When Python encounters a name in a program, it looks for that identifier in an expanding set of contexts:
  1. ***Local.*** The local context consists of all names defined within the current function. A name is defined in the function if it appears as a parameter, as the target of an assignment, as the index variable in a `for` loop, or as a nested function definition.
  2. ***Enclosing.*** The enclosing context consists of the names defined in a function that encloses the current one.
  3. ***Global.*** The global context consists of names defined outside of any function or imported into the current module using the `from-import` statement.
  4. ***Built-in.*** The last place that Python looks for a name is in the list of built-in functions like `abs`, `str`, and `print`.
- The part of a program in which a name is defined is called its ***scope***.

The `GWindow` Class Revisited

The following expanded set of methods are available in the `GWindow` class:

`add(`*object*`)`	Adds the object to the canvas at the front of the stack
`add(`*object*`,` *x*`,` *y*`)`	Moves the object to (*x*, *y*) and then adds it to the canvas
`remove(`*object*`)`	Removes the object from the canvas
`removeAll()`	Removes all objects from the canvas
`getElementAt(`*x*`,` *y*`)`	Returns the frontmost object at (*x*, *y*), or `null` if none
`getWidth()`	Returns the width in pixels of the entire canvas
`getHeight()`	Returns the height in pixels of the entire canvas
`setBackground(`*c*`)`	Sets the background color of the canvas to *c*.

The Two Forms of the `add` Method

- The `add` method comes in two forms. The first is simply

```
add(object)
```

which adds the object at the location currently stored in its internal structure. You use this form when you have already set the coordinates of the object, which usually happens at the time you create it.

- The second form is

```
add(object, x, y)
```

which first moves the object to the point (*x*, *y*) and then adds it there. This form is useful when you need to determine some property of the object before you know where to put it.

Methods Common to All `GObjects`

`setLocation(`*x*`,`*y*`)`	Resets the location of the object to the specified point
`move(`*dx*`,` *dy*`)`	Moves the object *dx* and *dy* pixels from its current position
`movePolar(`*r*`,` *theta*`)`	Moves the object *r* pixel units in direction *theta*
`getX()`	Returns the *x* coordinate of the object
`getY()`	Returns the *y* coordinate of the object
`getWidth()`	Returns the horizontal width of the object in pixels
`getHeight()`	Returns the vertical height of the object in pixels
`contains(`*x*`,` *y*`)`	Returns `true` if the object contains the specified point
`setColor(`*c*`)`	Sets the color of the object to the `Color` *c*
`getColor()`	Returns the color currently assigned to the object
`scale(`*sf*`)`	Scales the shape by the scale factor *sf*
`rotate(`*theta*`)`	Rotates the shape counterclockwise by *theta* degrees
`sendToFront()`	Sends the object to the front of the stacking order
`sendToBack()`	Sends the object to the back of the stacking order
`sendForward()`	Sends the object forward one position in the stacking order
`sendBackward()`	Sends the object backward one position in the stacking order

Additional Methods for `GOval` and `GRect`

Fillable shapes (`GOval` and `GRect` [and later `GArc` and `GPolygon`])

`setFilled(`*flag*`)`	Sets the fill state for the object (`False`=outlined, `True`=filled)
`isFilled()`	Returns the fill state for the object
`setFillColor(`*c*`)`	Sets the color used to fill the interior of the object to *c*
`getFillColor()`	Returns the fill color

Resizable shapes (`GOval` and `GRect` [and later `GImage`])

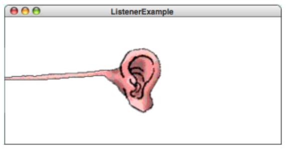
`setSize(`*width*`, `*height*`)`	Sets the dimensions of the object as specified
`setBounds(`*x*`, `*y*`, `*width*`, `*height*`)`	Sets the location and dimensions together

The Python Event Model

- Graphical applications usually make it possible for the user to control the action of a program by using an input device such as a mouse. Programs that support this kind of user control are called ***interactive programs***.
- User actions such as clicking the mouse are called ***events***. Programs that respond to events are said to be ***event-driven***.
- In modern interactive programs, user input doesn't occur at predictable times. A running program doesn't tell the user when to click the mouse. The user decides when to click the mouse, and the program responds. Because events are not controlled by the program, they are said to be ***asynchronous***.
- In Python program, you write a function that acts as a ***listener*** for a particular event type. When the event occurs, that listener is called.

The Role of Event Listeners

- One way to visualize the role of a listener is to imagine that you have access to one of Fred and George Weasley's "Extendable Ears" from the Harry Potter series.
- Suppose that you wanted to use these magical listeners to detect events in the canvas shown at the bottom of the slide. All you need to do is send those ears into the room where, being magical, they can keep you informed on anything that goes on there, making it possible for you to respond.



First-Class Functions

- Writing listener functions requires you to make use of one of Python's most important features, which is summed up in the idea that functions in Python are treated as data values just like any others.
- Given a function in Python, you can assign it to a variable, pass it as a parameter, or return it as a result.
- Functions that have are treated like any data value are called ***first-class functions***.
- The textbook includes examples of how first-class functions can be used to write a program that generates a table of values for a client-supplied function. The focus in today's lecture is using first-class functions as listeners.

Nested Functions and Closures

- Because functions are first-class values in Python, it must be the case that inner functions have values that can be assigned of passed as parameters.
- The value of an inner function consists of more than the code that implements it. Those values also include a reference to the outer context in which that function is defined.
- This combination of a function definition and the collection of local variables available in the context in which the new function is defined is called a ***closure***.
- Closures are essential to writing interactive programs in Python, so it is worth going through several examples in detail.

A Simple Interactive Example

- The first interactive example in the text is `DrawDots`:

```
def DrawDots():
    def clickAction(e):
        gw.add(createFilledCircle(e.getX(), e.getY(),
                                  DOT_SIZE / 2))

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    gw.addEventListener("click", clickAction);
```

- The key to understanding this program is the `clickAction` function, which defines what to do when the mouse is clicked.
- It is important to note that `clickAction` has access to the `gw` variable in `DrawDots` because `gw` is included in the closure.

Registering an Event Listener

- The last line in the **`DrawDots`** function is

```
gw.addEventListener("click", clickAction);
```

which tells the graphics window (**`gw`**) to call **`clickAction`** whenever a mouse click occurs in the window.

- The definition of **`clickAction`** is

```
def clickAction(e):
    gw.add(createFilledCircle(e.getX(), e.getY(),
                                DOT_SIZE / 2))
```

which uses the **`createFilledCircle`** function from the **`gtools`** library.

Callback Functions

- The **`clickAction`** function in the **`DrawDots.py`** program is representative of all functions that handle mouse events. The **`DrawDots.py`** program passes the function to the graphics window using the **`addEventListener`** method. When the user clicks the mouse, the graphics window, in essence, calls the client back with the message that a click occurred. For this reason, such functions are known as ***callback functions***.
- The parameter **`e`** supplied to the **`clickAction`** function is a data structure called a ***mouse event***, which gives information about the specifics of the event that triggered the action.
- The programs in the text use only two methods that are part of the mouse event object: **`getX()`** and **`getY()`**. These methods return the *x* and *y* coordinates of the mouse click in the coordinate system of the graphics window.

Mouse Events

- The following table shows the different mouse-event types:

`"click"`	The user clicks the mouse in the window.
`"dblclk"`	The user double-clicks the mouse.
`"mousedown"`	The user presses the mouse button.
`"mouseup"`	The user releases the mouse button.
`"mousemove"`	The user moves the mouse.
`"drag"`	The user drags the mouse with the button down.

- Certain user actions can generate more than one mouse event. For example, clicking the mouse generates a **`"mousedown"`** event, a **`"mouseup"`** event, and a **`"click"`** event, in that order.
- Events trigger no action unless a client is listening for that event type. The **`DrawDots.py`** program listens only for the **`"click"`** event and is therefore never notified about any of the other event types that occur.

A Simple Line-Drawing Program

Drawing a line requires three actions: pressing the mouse button at the start, dragging it to the end, and then releasing the mouse. The function **`mousedownAction`** creates a zero-length line that begins and ends at the current mouse position. Dragging the mouse results in a series of **`drag`** events in rapid succession. Each call resets the end point of the line. The effect of this strategy is that the user sees the line as it grows, providing the necessary visual feedback to position the line. As you drag the mouse, the line stretches, contracts, and changes direction. This technique is called ***rubber-banding***.

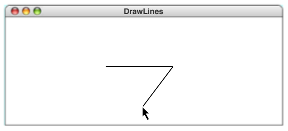
```
def DrawLines():
    def mousedownAction(e):
        nonlocal line
        line = GLine(e.getX(), e.getY(), e.getX(), e.getY())
        gw.add(line)

    def dragAction(e):
        line.setEndPoint(e.getX(), e.getY())

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    line = None
    gw.addEventListener("mousedown", mousedownAction)
    gw.addEventListener("drag", dragAction)
```

Simulating the `DrawLines` Program

– The two calls to **`addEventListener`** register the listeners.
– Depressing the mouse button generates a **`"mousedown"`** event.
– The **`mousedownAction`** call adds a zero-length line to the canvas.
– Dragging the mouse generates a series of **`"drag"`** events.
– Each **`dragAction`** call changes the end point of the line.
– Releasing the mouse disables the dragging operation.
– Repeating these steps adds new lines to the canvas.



The `nonlocal` Statement

- The **`DrawLines`** program contains the mysterious line

```
nonlocal line
```

- The purpose of this statement is to ensure that Python uses the definition of the variable **`line`** from the enclosing scope and does not create a new local variable with the same name.
- Python's scope rule indicates that "the local context consists of all names defined within the current function." The fact that **`mousedownAction`** contains an assignment to line makes the **`nonlocal`** declaration necessary.
- Chapter 5 offers a strategy for making **`nonlocal`** declarations unnecessary. I will use this strategy for the rest of the semester, but you need to understand what **`nonlocal`** means.