

CS I

Introduction to Computer Programming

Lecture 13: November 5, 2012

**Graphics 3: More
Event Handling**



Caltech CS 1: Fall 2012

Last time

- Graphics
 - lines, circles, ovals
 - event handling
 - **mainloop()**
 - the **bind** method
 - binding the keypress '**q**' to the function **quit**



Today

- More event handling
 - key presses, mouse clicks
 - using events
- Graphical object "handles"
 - A way to control objects already on the canvas
- Global variables



Recall from last time

- Last time we had:

```
from Tkinter import *
root = Tk()
root.geometry('800x600')
c = Canvas(root, width=800, height=600)
c.pack()
r = c.create_rectangle(0, 0, 50, 50,
                      fill='red', outline='red')
root.bind('<q>', quit)
root.mainloop()
```



Callbacks

```
root.bind('<q>', quit)
```

- Functions like **quit** which are arguments to the **bind** method are called event handlers
 - because they "handle events"
- Also often called callbacks or callback functions
- The binding object (here, **root**) stores them and "calls them back" whenever the event happens
- When it does this, it passes the event as the only argument to the function



Callbacks

- One negative thing about this:
- When **q** is pressed, the program exits
 - but before it does, it prints out something like:
<Tkinter.Event instance at 0x776058>
- We saw why last time
 - **Tkinter** passes the event object as an argument to the event handler
- Let's see how to improve this!



Callbacks

- We would like for the program to exit when the **q** key is pressed, without printing anything
- We can do this by defining our own callback function
 - and then binding it to the **q** key
- Let's change the code accordingly



Callbacks

- Our new callback function:

```
def exit_python(event):  
    '''Exit Python when the event  
    'event' occurs.'''  
  
    quit() # no arguments to quit
```

- Then, before the `mainloop` line, write:

```
root.bind('<q>', exit_python)
```

- instead of

```
root.bind('<q>', quit)
```



Callbacks

- Now, when we hit the `q` key, the program exits and nothing is printed
- The `exit_python` function received the event and ignored it
 - just calls `quit` without any arguments
 - so nothing is printed
- This is the behavior we wanted
- Now let's try some variations on this



Handling different events

- Change '<q>' to '<Key>':

```
root.bind('<Key>', exit_python)
```

- Now, the program will exit when any key on the keyboard is pressed, not just the **q** key

- Change '<Key>' to '<Button-1>'

```
root.bind('<Button-1>', exit_python)
```

- Now the program will exit when the left mouse button (called "Button-1" by **Tkinter**) is clicked



root vs canvas

- So far, we've been binding event handlers (callback functions) to the **root** object
- This is not the only **Tkinter** object that can handle events
- The **canvas** object can also handle events by itself
- Can try binding event handlers to the **canvas** object instead of the **root** object



root vs canvas

- Let's replace the line:

```
root.bind('<q>', exit_python)
```

- with

```
c.bind('<q>', exit_python)
```

- (**c** was the name of the canvas object)
- Run our program again...
- hit the **q** key, and...
- Nothing happens!



root vs canvas

- The canvas object doesn't handle key press events!
- Instead, it passes them on to its parent object (the **root** object)
- The point: graphical objects may not be able to handle every kind of event!
- However...



root vs canvas

- Let's replace the line:

```
root.bind('<q>', exit_python)
```

- with

```
c.bind('<Button-1>', exit_python)
```

- This will work!
- Canvas objects do handle mouse button events
- When working with graphical objects, need to know what events they can handle



Continuing...

- Now, we'll look at more interesting callback functions



Outline of our programs

- We're going to work through a few graphics programs
- Each will have the same overall structure
- Details will be different
 - drawing functions
 - callback functions



Outline of our programs

```
from Tkinter import *
import random
```

<drawing functions>

<callback functions>

```
if __name__ == '__main__':
    # Set everything up and go.
```



Outline of our programs

<... as before ...>

```
if __name__ == '__main__':
    root = Tk()
    root.geometry('800x600')
    canvas = Canvas(root, width=800, height=600)
    canvas.pack()
```

<... Code to set up event handlers.>

```
root.mainloop() # Start the event loop.
```



Program I: Overview

- This program will
 - quit when **q** is pressed
 - print out the (**x**, **y**) coordinates of the mouse cursor (relative to the canvas window) when the left mouse button is clicked



Program I: Last part

```
if __name__ == '__main__':
    <set up root and canvas>

    root.bind('<q>', exit_python)
    canvas.bind('<Button-1>', button_handler)

root.mainloop()
```



Program 1: Bindings

- `root.bind('<q>', exit_python)`
 - (`q` key calls the `exit_python` function)
-
- `canvas.bind('<Button-1>', button_handler)`
 - (Clicking left mouse button calls the `button_handler` function)



Program 1: Callback functions

```
def exit_python(event):
    '''Exit Python.'''
    quit()

def button_handler(event):
    '''Handle left mouse button click events.'''
    print 'x = %d, y = %d' % (event.x, event.y)
```



Program 1: Callback functions

```
def exit_python(event):  
    '''Exit Python.'''  
  
    quit()
```

Have seen this before

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
  
    print 'x = %d, y = %d' % (event.x, event.y)
```



Program 1: Callback functions

```
def exit_python(event):  
    '''Exit Python.'''  
    quit()
```

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    print 'x = %d, y = %d' % (event.x, event.y)
```

This is new



A mouse click event

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    print 'x = %d, y = %d' % (event.x, event.y)
```

- When you click on the mouse, **Tkinter** creates an *event object* that contains all the relevant data about the event
- We have bound this kind of event to the **button_handler** callback function



A mouse click event

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    print 'x = %d, y = %d' % (event.x, event.y)
```

left-mouse-button-click event object

- Tkinter calls **button_handler** with the event object as its only argument



A mouse click event

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    print 'x = %d, y = %d' % (event.x, event.y)
```

- The event object contains two useful pieces of data: `event.x` and `event.y`
- They represent the `x` and `y` coordinates of the mouse cursor when the left mouse button was clicked
 - Coordinates are in pixels, relative to the root window



Aside: Object attributes

- `event.x` and `event.y` are *attributes* of the `event` object
- Mostly we have been using *methods* of objects
- Methods are attributes that happen to be functions
- Objects can also have non-function attributes
- Here, `event.x` and `event.y` are both integers (coordinates in pixels)
 - relative to the origin of the canvas object



A mouse click event

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    print 'x = %d, y = %d' % (event.x, event.y)
```

- When the left mouse button gets clicked, this function will get the event object
- It will print out the **x** and **y** coordinates of the mouse cursor



Program I Demo

- See it in action...



Interlude

- Midterm!
 - Up this week, due next week
 - Cover sheet will have all details
 - Covers up to lab 4, this lecture
 - Worth 6.0 points (no min grading)
- Movie clip!
 - The "Good Old Days"



Moving on

- Now that we can capture the **x** and **y** coordinates of the mouse cursor, there are lots of things that we can do with this information
- This leads us to program 2
- Clicking on the mouse will draw a square at the **(x, y)** coordinate where the mouse cursor is
 - random color, random size
- Only need to make a couple of changes



Program 2

- Extra graphics command:

```
def draw_random_square(canvas, x, y, \
                       min_size, max_size):
    # ... code omitted ...
```

- This function takes
 - a canvas
 - an **(x, y)** pair of coordinates
 - a minimum and maximum size
 - and draws a randomly-colored square of a random size between the minimum and maximum values



Program 2

- Extra graphics command:

```
def draw_random_square(canvas, x, y, \
                      min_size, max_size):
    # ... code omitted ...
```

- We'll assume this has been defined
 - very similar to lab 4 code ;-)



Program 2

- Change **button_handler** callback function

```
def button_handler(event):
    '''Handle left mouse button click events.'''
    draw_random_square(canvas,
                        event.x, event.y,
                        50, 150)
```

- That's it for changes!



Program 2 Demo

- See it in action...



Global variables

- One subtle thing in program 2:

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    draw_random_square(canvas,  
                        event.x, event.y,  
                        50, 150)
```

- Where does the **canvas** come from?
 - Not an argument to this function



Global variables

- Later in the code we have:

```
root = Tk()  
root.geometry('800x600')  
canvas = Canvas(root, width=800, height=600)  
canvas.pack()
```

- This is the **canvas** that was being referred to
- It is a *global variable*
- That just means that it was defined outside of any function
 - sometimes referred to as the "top level" of the program



Global variables

- Python lets you use global variables
 - Most of the time, want to *avoid* using them
- A global variable can be changed by *any* function
 - so if it gets the wrong value assigned to it, it can be very hard to find out which function was responsible
- A variable defined inside a function is a *local variable*
- Local variables can *only* be changed inside the function they were defined in



Global variables

- Local variables are *much* safer than global variables
- Try to avoid using global variables whenever possible
- So...
 - why are we using one here?
 - Why not just pass the **canvas** object as another argument to the **button_handler** callback function?



Back to callback functions

- Answer: callback functions are required to take only *one* argument
 - which must represent the event that activates them
- This is a limitation of **Tkinter**'s design
- **Tkinter** doesn't know how many arguments a particular callback function might want in addition to the event
 - and even if it did, it wouldn't know what to put there
- So any additional data has to be passed via global variables (ugly)



Preview: classes

- There is a "nice" solution to this problem which does not require global variables
- Will involve Python **classes**
 - user-defined object types
- We will see much more about classes starting next lecture
- For now, we'll stick to using global variables
 - (and hold our noses)



Graphical object handles

- So far we've seen how to create graphical objects
 - directly as part of the program
 - through user interaction with callback functions
- However, once we've created an object, we haven't done anything with it
- It would be useful to be able to manipulate an object after it's created
 - to move it, change it in some way, delete it



Graphical object handles

- **Tkinter** has had this ability all along
 - We just haven't been using it!
- **Tkinter** canvas commands to create graphical objects return a value
- This value is called a **handle** to the graphical object
 - It is *not* the graphical object itself!



Handles and handlers

- Don't confuse *event handlers* (functions which are called when certain events happen) with canvas graphical object *handles*
 - (no relationship whatsoever between the two concepts)
- A handle is just an integer (a Python `int`) which represents a particular graphical object on a canvas



Handles and handlers

- Canvasses don't let you *directly* manipulate the objects they contain
 - have to do it *indirectly* through the object handles
- In other words, graphical objects on canvasses
 - like rectangles, ovals, lines, etc.
- are not exposed as Python objects
- The only way to do anything to them is through canvas methods that require handles as arguments



Example: deleting objects

- Our programs have created a bunch of graphical objects (squares) on a canvas
- After a while, a canvas can get very cluttered
- Might want to delete some or all of the existing squares
- Can do this easily using handles



Example: deleting objects

- We will extend our program so that pressing the **c** key (**c** for *clear*) will remove all the squares from the canvas



Revising our drawing functions

- Before we mentioned this function:

```
def draw_random_square(canvas, x, y, \
                       min_size, max_size):
    # ... code omitted ...
```

- Somewhere in this function there would need to be a call to

```
canvas.create_rectangle(...)
```

- Whatever this method returns, we ignore
- And **draw_random_square** doesn't return anything



Revising our drawing functions

- We will change this line to:

```
square = canvas.create_rectangle(...)
```

- And we will make `draw_random_square` return this value (a handle)
- **square** is just a Python `int`
 - not some kind of fancy Python object with attributes etc.
- For instance, if it was 5, that would mean "the 5th object created on this canvas"
- Properties of the graphical object are stored *inside* the canvas



Storing object handles

- OK, so `draw_random_square` returns a handle to a square object on a particular canvas
- We want to store this so we can delete the square later on
- We will create a (global) list of squares and add every new square to the list



Storing object handles

- At the end of the program:

```
if __name__ == '__main__':
    root = Tk()
    root.geometry('800x600')
    canvas = Canvas(root, width=800, height=600)
    canvas.pack()
    squares = []

    <rest of code as before>
```

- Now we can store all handles to squares in the (global) list called **squares**



Storing object handles

- Modify the **button_handler** callback function:

```
def button_handler(event):  
    '''Handle left mouse button click events.'''  
    square = draw_random_square(canvas,  
                                 event.x, event.y,  
                                 50, 150)  
  
    squares.append(square)
```

- All newly-created handles to squares are appended to the global **squares** list



Storing object handles

- Change the callback function for key presses:

```
def key_handler(event):  
    '''Handle key press events.'''  
  
    key = event.keysym  
  
    if key == 'q':  
        quit()  
  
    elif key == 'c':  
        for square in squares:  
            canvas.delete(square)
```

```
# Later in the code:  
root.bind('<Key>', key_handler)
```



Key events

```
root.bind('<Key>', key_handler)
```

- This line says: every time a key is pressed on the keyboard, call the `key_handler` callback function
 - passing it the `event` object corresponding to the key press event



Key events

- The key handler callback function:

```
def key_handler(event):  
    '''Handle key press events.'''  
    key = event.keysym  
  
    ...
```

- Key press event objects have an attribute called a **keysym**
- This is basically just the character on the key that was pressed
 - so '**q**' for the **q** key, '**c**' for the **c** key, etc.



Key events

```
def key_handler(event):
    '''Handle key press events.'''
    key = event.keysym
    if key == 'q':
        quit()
    elif key == 'c':
        for square in squares:
            canvas.delete(square)
```

- This makes the **q** key quit Python as before



Key events

```
def key_handler(event):
    '''Handle key press events.'''
    key = event.keysym
    if key == 'q':
        quit()
    elif key == 'c':
        for square in squares:
            canvas.delete(square)
```

- This makes the **c** key delete all the squares from the canvas
 - using the **canvas.delete** method



Program 3 Demo

- See it in action...



Modifying global variables

- One problem with this approach:
- The squares in the **squares** list were deleted using the **canvas.delete** method
- But the **squares** list itself wasn't emptied out
- Let's try to do that now



Modifying global variables

```
def key_handler(event):
    ...
    elif key == 'c':
        for square in squares:
            canvas.delete(square)
    squares = []
```

- Here, we're trying to change the global variables **squares** by assigning the empty list to it
- Only one problem: it won't work!



Modifying global variables

```
def key_handler(event):
    ...
    elif key == 'c':
        for square in squares:
            canvas.delete(square)
    squares = []
```

- Python has no way of knowing that **squares** is supposed to represent a global variable
- It could just as well be a *local* variable that we just decided to create at that point in the function



Modifying global variables

```
def key_handler(event):  
    ...  
    elif key == 'c':  
        for square in squares:  
            canvas.delete(square)  
        squares = []
```

- When in doubt, Python assumes that a variable is a local variable, not a global variable
- How do we tell it "**squares** is supposed to be a global variable, not a local variable"?



Modifying global variables

```
def key_handler(event):
    global squares
    ...
    elif key == 'c':
        for square in squares:
            canvas.delete(square)
        squares = []
```

- The **global** declaration tells Python that **squares** refers only to a global variable inside the **key_handler** function
- Now, the global **squares** list will be cleared every time the **c** key is pressed



Modifying global variables

- A Python program can have the same name bound to both a local and a global variable
- When this happens, the local variable takes precedence (it "shadows" the global variable)
- Try hard to avoid this situation, though!
 - Very confusing to anyone reading your code!



Modifying global variables

- Rules of thumb:
- If you want to access a global variable inside a function...
 - and there is no local variable with the same name...
 - just access it as usual
- If you want to *modify* a global variable inside a function...
 - you have to declare it as global using the **global** declaration
- It's a pain to remember to do this, so avoid using global variables
- Classes (next lecture) are a better alternative



Next time

- Creating your own objects
- Python **classes**

