Lecture 12:  July 19, 2012
Graphics and Graphical User Interfaces (part 2)
This lecture

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

- Last time we had:
  ```python
  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.mainloop()
  ```
We added the `root.mainloop()` line in place of the `raw_input` line.
The drawing doesn't change.
Now, the only way to exit the program is to close the window or to quit Python.
So far, haven't added any code to handle any events.
Event 1: q for quit

- Let's add an event so that pressing the q key on the canvas quits the application.
- We will add just one line before the `mainloop` line:
  ```python
  root.bind('<q>', quit)
  ```
- Let's look at this line in detail.
Event 1: q for quit

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

- This says:
- The graphical object that will handle the event is the `root` object
- The event to be handled is when the user presses the `q` key on the keyboard
  - `Tkinter` represents this as the string `'<q>'`
- When the `q` key is pressed, the built-in `quit` function will be executed
  - This causes Python to exit immediately
Event 1: q for quit

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

- We say that this method call `binds` the event (pressing the `q` key on the keyboard) to the function `quit` (which handles the event)
Event 1: `q` for quit

- This works, but there is one odd thing
- When `q` is pressed, the program exits
  - but before it does, it prints out something like:
    `<Tkinter.Event instance at 0x776058>`
- What does this mean?
Event 1: ṣ for quit

- The `quit` function usually gets no arguments.
- If it gets an integer argument, it passes it to the operating system.
  - (the reason isn't important)
- If it gets a non-integer argument, it prints it and then exits.
- Somehow, here it got an argument of type `Tkinter.Event`.
Event 1: q for quit

- What happened:
- When the q key on the keyboard was pressed
  1. Tkinter created a Tkinter.Event object
  2. Tkinter's event loop (mainloop) checked to see if the q key was bound to any function
  3. It found that q was bound to the quit function
  4. It called quit with the Tkinter.Event object as its argument
  5. quit printed the event and then made Python exit
Event 1: \texttt{q} for quit

- Another weird thing...
- Look at the line:
  \begin{verbatim}
  root.bind('<q>', quit)
  \end{verbatim}
- \texttt{quit} is the name of a Python function
- The \texttt{bind} method of the \texttt{root} object can take a \texttt{function} as an argument!
- Python allows functions to be treated as data
  - functions are "first-class"
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 root object 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

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

- Then, before the `mainloop` line, write:
  ```python
root.bind('<q>', exit_python)
```
- instead of
  ```python
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>':
  ```python
call.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>':
  ```python
call.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:
  ```python
  root.bind('<q>', exit_python)
  ```
- with
  ```python
  c.bind('<q>', exit_python)
  ```
- (```c``` was the name of the canvas object)
- Run our program again...
- hit the `q` key, and...
- Nothing happens!
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

```python
from Tkinter import *
import random

<drawing functions>
<callback functions>

if __name__ == '__main__':
    # Set everything up and go.
```
Inside a module, the variable `__name__` is set to be
- the name of the module (when the module is imported)
- the special name `'__main__'` (when the module is run directly by Python)

Let's see how this works
Here's a simple module: `example.py`

```python
# module: example.py
print 'My name is: %s' % __name__
```

- It defines a Python module called `example`
- When it's imported, this code will be run
- Let's import it from other Python code:

```python
>>> import example
My name is: example
```

- Nothing strange so far...
What happens when the file `example.py` is executed directly by Python instead of imported?

- e.g. at the terminal command line

```bash
% python example.py
My name is: __main__
```

- This says: `example.py` is the first thing executed by Python
```python
if __name__ == '__main__':
    # example.py
    if __name__ == '__main__':
        print 'This is the main module.'
```

- We can use this to define some code in a module which only executes if the module is the first thing Python executes:
if __name__ == '__main__':

- Now, at the terminal command line:

```python
% python example.py
This is the main module.
```
if __name__ == '__main__':

- But when importing the module in the Python shell:

  >>> import example

- Nothing happens!
- Can use this trick to execute some code only when a module is loaded as the main module
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 1: Overview

- This program will
  - quit when q is pressed
  - print out the (x, y) coordinates of the mouse cursor (relative to the window) when the left mouse button is clicked
Program 1: Last part

```python
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

```python
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)
```

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

- 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).
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 1 Demo

- See it in action...
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:

```python
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:
  ```python
def draw_random_square(canvas, x, y, 
min_size, max_size):
    # ... code omitted ...
  
  We'll assume this has been defined
Program 2

- Change `button_handler` callback function

```python
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:

```python
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:

```python
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 `draw_random_square` callback function?
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 later in your life
- 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 which represents a particular graphical object on a canvas
Handles and handlers

- Canvases 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 canvases
  - 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:
  
  ```python
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:
  \[\text{square} = \text{canvas}.\text{create\_rectangle}(\ldots)\]
- And we will make `draw\_random\_square` return this value
- `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 5\(^{th}\) object created on this canvas"
- Properties of the object 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:

```python
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:

```python
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:

```python
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:

```python
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.
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
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
def key_handler(event):
    
    # Code for handling different key events...

    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!
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
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"?
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** array will be cleared every time the **c** key is pressed
Modifying global variables

- The `global` declaration is only needed if you intend to change what the global variable name is bound to (i.e. to a new object).
- If you just want to do something with it (like append something to a global list) you don't need the `global` declaration.
- So we didn't need it in `button_handler` when we had the line:
  ```python
  squares.append(square)
  ```
- because `squares` is still the same list object.
Next lecture

- More graphics
- Widgets!
  - buttons, menus, etc.