

3 Appendix: Python GUI Development

Structure of Python Applications

Python is an object-oriented language, if you choose to use it that way. Object-oriented design is an effective method of organizing applications. At the top level, we'll create a class that represents the application. All of the capabilities of the application will be methods of the application class. The application's `init` method will handle program startup, and its main method will begin the event loop that waits for user input.

Elements of a GUI:

- **Root:** the base object for a Tkinter application. It is the meta-manager for windows, menus, and events. In Tkinter, the root can also be attached to a visible window, but you can make the root window invisible and create other windows that are separate from the root window.
- **Windows:** visual window elements for the application. Windows are containers for the Canvas and other GUI elements like buttons and sliders.
- **Canvas:** a virtual drawing space on which visual elements, graphics, and images reside. The Canvas is separate from, and can be much larger than the window. Only the part of the Canvas within the window can be seen.
- **Menus:** drop-down or pop-up lists of commands a user can select using a mouse.
- **Events:** a keypress, mouse click, mouse drag, menu selection, or other events such as a window coming into the foreground.
- **Callbacks:** a function that is called when an event occurs.

3.0.3 Application Design

Packages: The packages we will need include the Tkinter and `tkFileDialog` packages. We may add others as we add capabilities. The Tkinter package is the general package for windows, menus, events and other widgets. The `tkFileDialog` handles giving the user a standard file selection dialog.

Application class: A good design strategy is to encapsulate your application as a class. The `init` method for the class sets up the application, and all of the callbacks and handler functions are methods. All of the application state information is stored as fields of the application class, which guarantees that all class methods have access to application state data. If you don't use a class, you'll end up creating a class to hold that information anyway.

Initialization: The `__init__` method has to do a number of things to set up the application.

- Create a root widget
`self.root = Tkinter.Tk()`
- Set up the geometry of the window
`self.root.geometry( "%dx%d+50+30" % (dx, dy) )` (note the X-window geometry)
- Set the window title
`self.root.title( "Display App" )`
- Set the maximum size of the window for resizing
`self.root.maxsize( 1024, 768 )`

- Bring the window to the front
`self.root.lift()`
- Make the menus (write a function to handle it)
- Create the Canvas widget (write a function to handle it)
- Set the key and button bindings (write a function to handle it)
- Set up any state information required by the application (write a function to handle it)

Menus: The menus are set up as a tree. At the root is the menu bar. The main pull-down menus are children of the menu bar, and fold-out menus can be inserted into them. All of the elements of the tree are Menu objects, even the menu bar. The following sequence creates the menu bar, sets it as the main menu widget for the Tk root, then creates a file menu and a command menu and adds them into the menu bar.

```
# create a new menu
self.menu = tk.Menu(self.root)

# set the root menu to our new menu
self.root.config(menu = self.menu)

# create a file menu
filemenu = tk.Menu( self.menu )
self.menu.add_cascade( label = "File", menu = filemenu )

# create a command menu
cmdmenu = tk.Menu( self.menu )
self.menu.add_cascade( label = "Command", menu = cmdmenu )
```

To add items to a menu we use the `add_command` and `add_separator` methods. For example, to build a file menu with open and quit options, we could write the following. The calls to `add_command` tell Tk to call the `handleOpen` and `handleQuit` methods when the user selects Open and Quit, respectively.

```
filemenu.add_command( label = "Open...", command = self.handleOpen )
filemenu.add_separator()
filemenu.add_command( label = "Quit", command = self.handleQuit )
```

Canvas: The Canvas object provides the drawing surface on which we create widgets (controls) and visual elements such as lines, dots, circles, etc. If we have no other objects to create, then all that needs to occur for initialization is creating and packing the canvas. The Canvas size should, in general, be at least as big as the window.

```
self.canvas = tk.Canvas( self.root, width=dx, height=dy )
self.canvas.pack( expand=tk.YES, fill=tk.BOTH )
```

Bindings: A binding connects a user action, such as a mouse button click or keypress, to a specific function. Almost any user action can be bound to a function. In the example below, the left mouse button (button 1) and motion while the left mouse button is held down are bound to two separate functions.

```
self.root.bind( '<Button-1>', self.handleButton1 )
self.root.bind( '<B1-Motion>', self.handleButton1Motion )
```

Functions that handle events must take an event object as a parameter. The event object has many useful fields that provide information about the event. The method `handleButton1`, for example, might look like the following.

```
def handleButton1(self, event):  
    print 'handle button 1: %d %d' % (event.x, event.y)
```

You can bind functions to regular keyboard events as well as events where the user is holding down a modifier key. For example, to catch the event where the user holds down the command key and the q key, you would make the following binding.

```
self.root.bind( '<Command-q>', self.handleModQ )
```

You can also use the terms `Option-q` and `Control-q` for the option and control keys, respectively.

Drawing Objects: Tkinter allows you to make simple graphical elements on a Canvas. Standard Canvas graphical objects include arcs, images, lines, ovals, polygons, rectangles, and text. Each of these objects has many options that control its appearance. As an example, the following creates a plus sign around a given point using two lines. Note that the code stores a reference to each line created in the list `self.lines`. If you want to edit or change the appearance of objects, you have to store a reference to the object. It's useful to initialize whatever list or data structure you use to store graphical objects in the initialization function.

```
curline = self.canvas.create_line( event.x-5, event.y, event.x+5, event.y )  
self.lines.append(curline)  
curline = self.canvas.create_line( event.x, event.y-5, event.x, event.y+5 )  
self.lines.append(curline)
```

If we keep around the references to all of the lines we create, it means we can later move them using the `coords` method or delete them when we're done. The following example shows how you would traverse the list and delete all of the line elements.

```
for line in self.lines:  
    self.canvas.delete( line )
```

Moving Objects: Tkinter provides a function `coords` that lets you both query and move objects. If you use the function with only the graphics object as the argument, it returns the coordinates of that object. If you use the function with the graphics object and new coordinates as arguments, it moves the object.

```
# get the coordinates of the line  
loc = self.canvas.coords( line )  
  
# shift the line two pixels down and to the right  
self.canvas.coords( line, loc[0]+2, loc[1]+2, loc[2] + 2, loc[3] + 2 )
```

Window Coordinates: The coordinate system of the window has its origin in the upper left corner, with positive X going right and positive Y going down. This configuration is slightly different than we're used to thinking in that Y increases as you move down instead of up. It's an artifact of the fact that cathode ray tube displays were designed to draw pixels on the screen beginning in the upper left and ending in the lower right.