Lab Exercise 9

Software Design **Allen B. Downey**
Spring 2008

Reading: Chapter 19 of *How to think...*
Due: never!!!

9.1 GUI

1. Download widget_demo.py and `HelloGui.py` from the usual place and run them. Read the code and make sure you understand how it works. widget_demo.py demonstrates the basic use of widgets.

   `HelloGui.py` demonstrates the usual structure of an interface that inherits from `Gui`. It also displays an object diagram and a class diagram that demonstrate the relationships among the objects and classes you will be working with.

2. Download `MyWorld.py` from the usual place. It is an example of how you can use inheritance to customize the behavior of existing classes without having to modify the original. This module contains three class definitions:

   MyWorld: extends `TurtleWorld` and overrides `setup`, allowing you to customize the Gui features for the World.

   MyTurtle: extends `Turtle`, although at the moment it doesn't override anything.

   MyTurtleControl: extends `TurtleControl` and overrides `setup`, again so you can customize the Gui features for the turtle.

   As usual, you should read over the code and make sure you understand it.

3. The purpose of this lab exercise is to encourage you to play around and get familiar with the GUI elements in Tkinter.

   You can also experiment with rearranging the widgets, but for now you probably don't want to spend much time adjusting their appearance.

   Although Tkinter and my interface to it, `Gui.py` are relatively easy to use, compared to most window toolkits, the documentation can be hard to work with. Before you start, you might want to skim

   (a) `Gui.html` and `World.html`, both of which are included with Swampy.

   (b) The semi-official Tkinter reference, *An Introduction to Tkinter* at `http://www.pythonware.com/library/tkinter/introduction/`.

   You might find it useful to try the following exercises, but you don't have to do them all, and you are welcome to try other things.

   (a) Add a button to `MyWorld.setup` that draws an item on the canvas when it is pressed.

(b) Copy the `draw` method from `Turtle` into `MyTurtle` and then modify it so that your Turtles look different from those plain-belly turtles.

(c) Add a new method to the `MyTurtle` class, like `dance`, and then add a button to the `MyTurtleControl` panel so that when the button is pressed, it invokes the new Turtle method.

(d) While you are at it, modify your Turtles so that their "pen color" is their current color.

(e) Add an entry to the Turtle control panel and put the integer 1 into it. Then modify the Turtle so that when it moves, it draws a line with the line width specified in the entry.

(f) Add up and down buttons that increment and decrement the contents of the line width entry.

(g) Replace the pen up and pen down buttons with a single check button that controls whether the pen is down.

(h) Replace the menubutton that controls the turtle color with a sequence of radiobuttons or a listbox.

4. Download `HelloGui2.py` from the usual place and run it. Read the code and make sure you understand how it works. It demonstrates canvas items, tags, binding and drag-n-drop. We will talk about events and bindings more in class.

   One of the drawbacks of object-oriented code is that it is often difficult to follow the flow of execution through a class hierarchy, especially if there are many methods with the same name. For example, when you invoke `Item.coords`, it invokes `GuiCanvas.coords`, which invokes `Canvas.coords` (which is part of Tkinter). This kind of design is good in the sense that it hides details, but it is bad in the sense that it is harder to find your way around in a system like this.

   UML diagrams are sometimes helpful for figuring this stuff out. If you use emacs, you might also find the IM-Python menu helpful; it gives you a summary of the classes and methods in the file.

5. Check out `Gui.py` and the Tkinter documentation to make sure you understand how coordinates are handled. Tkinter works in what I will call "pixel coordinates", which means that the origin is in the upper-left, the y-axis points down, and one unit corresponds to one pixel.

   Gui.py and HelloGui.py work in "canvas coordinates", which means that the origin is in the middle, the y-axis points up, and a unit is not necessarily one pixel (although that is the default). The methods `GuiCanvas.trans` and `GuiCanvas.invert` translate from canvas coordinates to pixel coordinates and back.

   The other abstraction provided by `Gui.py` is translation from "structured coordinates" to "flat coordinates." Gui.py works in paired coordinate lists, like `[[x1, y1], [x2, y2]]`. Tkinter works with a flattened list like `[x1, y1, x2, y2]`. The functions `pair` and `flatten` (in Gui.py) convert from flat lists to pairs and back.

   Part of the reason I am showing you all of this is that it is an example of the kind of multi-level design I want you to do in your project.

6. Your mission, should you choose to accept it, is to add "handles" to the canvas items. A handle (in this context) is a small canvas item that you use to control the coordinates (or

other features) of another item. For example, a circle item might have two small rectangles you can use to change the position and size of the circle.

Here are some intermediate steps you might want to implement and test:

(a) Change `Item.select` and `Item.release` so that they save and restore the original color of the object (rather than making everything blue).

(b) Add a `rectangle` method to `Hello` and add a Rectangle button to the GUI.

(c) Add a `draw_handle` method to the `Item` class that gets the coordinates of the item and draws a handle at each coordinate.

(d) Modify `Item.select` so that when the user clicks on an item, the handles appear.

(e) Define a new class `Handle` that inherits from `Item`. For each handle item, create a Handle object.

(f) Add methods to the `Handle` class to override the inherited behavior appropriately. For example, Handles should not have handles of their own, but they should have drag-n-drop behavior.

(g) Modify `Item` so that when an Item moves, the Handles move, too.

(h) Modify `Handle` so that when a Handle moves, it moves the corresponding coordinate of the related Item.

You probably won't be able to get all of this done in lab, but you should do as much as you can. If you need capabilities like this for your project, this lab should get you off to a good start.

NOTE: the way `Gui.py` handles coordinates is not as polished as the rest of the program. If you run into difficulties, let me know!