Homework 1

Software Design **Allen B. Downey**
Spring 2007

Due: Wednesday 31 January at the beginning of lab.

The reading for this assignment is Chapters 2 and 3 of *How to think...*, so you might not be able to get through it in today's lab. Instead, let's get started, and we'll debug any problems. If you're not able to do the whole thing now, you will be after Monday's class.

1.1 World.py

1. If you have already created a user account in your Linux partition, log into it. Otherwise, follow the instructions in Chapter One of the Olinux Manual.
2. Once you are logged in as a user (not as root), create a terminal by right-clicking the mouse anywhere on the background and selecting Open Terminal. A window should appear with a prompt that ends in `$`.
3. If you type `pwd`, Linux should tell you that you are in your home directory.
4. Swampy is a suite of programs I wrote that we will use in this class. You can download it from the class web page by typing:

   ```
   wget http://wb/sd/code/swampy.1.0.zip
   ```

   A zip file is an archive that contains multiple files and sometimes directories. You can unpack it by typing:

   ```
   unzip swampy.1.0.zip
   ```

   The result should be a directory named `swampy.1.0`. I am going to recommend that you write most of your programs in this directory, so to eliminate some typing (and confusion), you should rename the directory using `mv`, which stands for "move":

   ```
   mv swampy.1.0 sd
   ```

   And then change directories with `cd`:

   ```
   cd sd
   ```

   Type `ls` to see the list of files in this directory. These files are a work in progress. They contain some code we will use this semester, and some code that is just there to demonstrate various features, some old code that doesn't do anything, and probably lots of good bugs. Over the course of the semester, we will look over this code and you will have the chance to improve it if you are so inclined.

5. Run TurtleWorld by typing `python World.py`. A window should appear with a (blank) canvas on the left and buttons on the right.

6. Press the Make Turtle button. A turtle should appear on the canvas and a Turtle Control panel should appear on the right. Push the buttons in the Turtle Control panel to move the turtle around, raise and lower the pen, and change the color of the turtle. Here are the names of the buttons and what they do:

```
fd      Move forward.
lt      Left turn.           rt      Right turn.
pu      Lift the pen.        pd      Lower the pen.
```

   There is a text entry between `bk` and `fd` that controls how far the turtle moves when either button is pushed. This value is an **argument** for the `fd` (`or bk`) function. Change the argument and press `fd`.

7. Press the Run code button. It executes the code in the text field, which should include the statements `world.clear()` and `bob = Turtle(world)`. The first statement erases all the turtles; the second line creates a new turtle named `bob`.

   Notice that there is no Turtle Control panel for `bob`. In order to control `bob`, you have to write a program.

8. Add a line of code to the program in the text window. To move the turtle, try something like `fd(bob, 90)`. The function `fd` takes two arguments, the name of a turtle and the distance you want the turtle to move. The other turtle control functions are:

```
bk(turtle, distance)           # move a turtle backward
lt(turtle)                     # turn left
rt(turtle)                     # turn right
pu(turtle)                     # pen up
pd(turtle)                     # pen down
```

9. Make a few errors. If the code you type in the text field contains an error, you should get an error message in the window you used to run the program. The message contains information about what was happening when the error occurred, most of which won't make sense to you. For example, if you spell `world` wrong, you'll get something like this:

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "/usr/lib/python2.4/lib-tk/Tkinter.py", line 1345, in __call__
    return self.func(*args)
  File "World.py", line 87, in run_text
    self.inter.run_code(source, '<user-provided code>')
  File "World.py", line 110, in run_code
    exec code in self.globals, self.locals
  File "<user-provided code>", line 2, in ?
NameError: name 'worl' is not defined
```

The last two lines are probably the most useful. The error occurred in line 2 of the "user-provided code" (that means you). It was a NameError; specifically, the name `worl` is not defined.

In general, error messages are a mixed blessing. They often contain information that helps you identify the problem, but they also contain information that is extraneous at best and misleading at worst. When you are starting out, it is a good idea to make errors on purpose so that you can see what the messages look like. Try some of the following:

(a) Leave out one of the parentheses in the function call.

(b) Put a semi-colon at the end of a line `;)`

(c) Change `Turtle` to `turtle` or `bob` to `Bob`. Yup, Python is case-sensitive.

10. You can define functions in the text field, too. At this point you might want to quit the program and run it again, so we start fresh.

    Now add the following lines to the text field (after the existing lines):

```
def fdlt(turtle, n):
    fd(turtle, n)
    lt(turtle)
```

    Read the program carefully and make sure you understand what it does. Now run it. Did you get what you expected? Reminder: the `def` statement only creates a new function; it doesn't execute it. To run the new function, you have to invoke (or call) it.

11. Add a line of code that invokes `fdlt`, passing `bob` as the first argument and a pleasant distance like 90 as the second. Run the program. What happens if the function call comes before the `def` statement?

12. The text field is convenient for typing and running a few lines of code, but it has the annoying property of vaporizing your code when the program quits. For longer (and longer-lived) programs, it would be better to put the code in a file.

    Use emacs to create a file named `turtle_code.py`. Copy and paste the code from the text field into the file. Save the file and then press the Run file button. It should execute your code.

13. In `turtle_code.py`, add a function called `ell` that takes a turtle and a distance as parameters and draws an ell-shape by invoking `fdlt` twice. Add a line of code that invokes `ell` (and remove the old line that invoked `fdlt`). Save the modified version of `turtle_code.py` and run it (you don't have to restart `World.py`).

14. Add a function called `square` that draws a square by invoking `ell` twice. Test your function.

15. CHECKPOINT #1: This is the end of what I hope you will be able to get done during the scheduled lab. If you have not read Chapter 3 yet, you should do that before you go on.

    If you are confident about what you are doing, you can go on to the next part of the homework. If not, now would be a good time to show what you have done to me or a course assistant. If you are off track, we can give you feedback that might save you a lot of time!

16. Create a file named `hello.py` and write a program that spells the letters `Hello` on the canvas. You can write the letters in any style you like; feel free to embellish them. More important than the style of the letters is the style of the code! A good solution to this problem should define and use functions that are well-named, demonstrably correct, appropriately general, and reusable. Ideally, your solution should be flexible, so that the size of the letters can be controlled by a parameter.

    For each function you write, add a comment that explains concisely what the function does. An important piece of information to document is where the turtle ends up at the end of the function.

    Hints:

    - You might find it useful to use more than one turtle.
    - You can speed up a turtle, or slow it down, by setting its `delay` attribute:

      ```
      bob = Turtle(world)
      bob.delay = 0.2          # time in seconds between moves
      ```

    - You might want to start by writing long, repetitive code, and then look for recurring idioms that would make good functions.
    - There is a tradeoff between writing lots of special-purpose functions and writing lots of repetitive code. Your goal should be to find a balance that yields reasonably concise, readable code. To evaluate your code, think about how you would handle the other letters of the alphabet. Do your functions lend themselves to reuse?

17. CHECKPOINT #2: This is the end of the required part of this homework. Please review the program in `hello.py` and clean it up: remove any unnecessary code, use white space to improve readability, and make sure that your comments are complete but concise. Add a comment at the beginning that has your name in it.

    By the time this homework is due, we will set up printing and I will give you more instructions on how to turn it in.

1.2 More fun

The following exercises are optional. You don't have to turn them in, but if you do something cool, I would be happy to see it.

1. To change the color of a turtle, try:

   ```
   turtle.set_color('papaya whip')
   ```

   The colors python knows about are in the file `/usr/X11R6/lib/X11/rgb.txt`

2. Use emacs to edit the file `World.py`. Under the IM-Python menu is a list of the classes defined in this file. The `Turtle` class contains a list of functions that define the behavior of turtles. Select the `fd` function in the `Turtle` class. Change `fd` so that when turtles draw lines, the lines are the same color as the turtles. Modify `hello.py` so that each letter is a different color.

3. See if you can figure out how to change the thickness of a line. Hint: type `man canvas`..

4. Go to the function `Turtle.draw` and see if you can figure it out. Can you make it draw a two-headed turtle? A turtle with a square shell? A turtle with thick legs? Or red legs? Or a really big turtle? Or a turtle that gets bigger depending on how far it is from the center of the screen?

5. Type the following code in the text field (or in `turtle_code.py`):

```
def hello():
    bob = Turtle(world)

world.bu(text='hello', command=hello)
```

   Run it. Does it make your head hurt?

6. Try this code:

```
world.bu(text='Run', command=world.run)
world.bu(text='Stop', command=world.stop)
```

   In the next week or two we will start animating turtles.

7. Look over the rest of `World.py` and see how much sense you can make of it. As I wrote this program, I tried to demonstrate a variety of Python features, and also many of the design patterns we will be talking about this semester.