CSC110 Supplemental Reading: Week04

Contents

1 GUI documentation

We are going to be using a custom module (not part of the standard Python library) in order to write programs that draw different shapes (circles, squares, etc) The module is called gui, and you must download this file, Gui.py in order to be able to use it.

I won't be teaching everything you need in order to be able to write a program from scratch. Instead you will get starter code and then we'll go over the syntax to be able to use certain features. NOTE: the Gui.py file must be in the same folder as the starter code.

1.1 The Drawing Area

In the starter code, the variable to access the area on which your program draws is called **canvas** (this is an object in our program). The canvas has a coordinate system just like you're used to: the origin is in the center of the canvas. If you move up the canvas, this is the positive y direction; move to the right on the canvas, that is the positive x direction.

The operations that belong to an object are called methods, instead of functions. We need to use new syntax to call methods: ***objectName.methodName()***

The methods below belong to the canvas object; therefore, we call them using the syntax: `canvas.methodName()`

- `clear()` – erases all the drawings from the canvas
- `config()` -- configures the canvas. You use this method to change the background color. You need to use a keyword argument to call config(). The one parameter that we'll use is the **bg** parameter (for background). If we want to set the background of the canvas to blue, the syntax is: **canvas.config(bg = 'blue')**

1.2 Drawing Shapes on the canvas

You will use the methods below to draw different shapes on the canvas. Each method has some required arguments, listed first, and then some optional keyword arguments, noted by the ***.

- `circle(center, radius, ***)`
- `rectangle(points, ***)`
- `line(points, ***)`
- `polygon(points, ***)`

Each method requires one or more points. A point is an x, y pair. In math you're used to writing them as (x, y). In Python, the syntax is to put the coordinates in square brackets. For example: **[3, 4]** (This is known as a list. We'll learn more about lists later this quarter)

If a method requires more than one point, then those points must be a list. For example, here we have a list of 3 points: **[[0,0], [10,17] , [2,3]]** Some of the methods below may also have the following optional parameters. I say optional because these parameters have a default value so you don't have to specify them. If you want to change the default value, you must then use keyword arguments:

- A **fill** parameter, specifying if the shape should be filled in with a color. The color options are:

  'white' 'black' 'red' 'green' 'blue' 'cyan' 'yellow' 'magenta'

  For example, you can set the fill of a shape as follows: **fill = 'blue'**. If you don't specify this parameter, the default is not to fill the shape in, but draw it as an outline. Even when a shape is colored in, there is still an outline in black.

- A **width** parameter, specifying how thick to make the border of the object (or, when drawing a line, how thick to make the line)

- An **outline** parameter where you can specify the color of the outline.

All of the numeric values (coordinates, radius) must be integers.

1.3 Method descriptions

`circle(center, radius, ***)` - draws a circle

> **center**: a point specifying the center of the circle
>
> **radius**: a length

`rectangle(points, ***)` – draws a rectangle

> The first parameter is a list of two points. The first point is the lower left corner of the rectangle. The second point is the upper right corner.

`line(points, ***)` – draws lines connecting the points

> The first parameter is a list of points. A line segment is drawn between each point (but no connecting the last point to the first point)

`polygon(points, ***)` – draws a polygon with the given points as vertices

> The first parameter again is a list of points (like line). The significant difference between this function and the line function is that the last point is connected to the first point to complete the shape. You can choose to fill this in or leave it as an outline.

1.4 Examples

Here are some examples of how to call these methods. Remember, you can't just type these into the Python shell; you must download the Gui.py file and the starter code and type these statements into the starter code.

```
canvas.circle([10, 10], 25, fill = ‘red’, width = 3)

canvas.rectangle( [ [20, 30], [70, 32] ], outline = ‘magenta’)

canvas.line([ [1, 1], [20, 20], [40, 1] ], width = 5)
```

Realize that shapes are drawn in the order the methods are called. If there is overlap of the shapes, the latter shape will be drawn on top of the earlier shape.

2 Functional Decomposition

The ability to write programmer-designed functions provides the foundation for a concept called **modular design, functional decomposition** or **procedural decomposition**. Several benefits are provided by the use of modular design and programmer-designed functions:

- **Code is easier to design** -- a large project can be broken down into smaller pieces, then each piece can be designed as a separate function. First identify the large tasks (such as INPUT, PROCESS, and OUTPUT), break these down until they are manageable units, then focus on designing each task as a function.

- **Code is easier to write** -- Once you have these individual functions designed, you can focus on writing just one at a time, instead of tackling the whole program at once. You can also give the task of writing different functions to different members of a programming team.

- **Code is easier to read** -- when functions are given meaningful names, the code that makes use of them can be almost self-documenting. In addition, detail can be hidden that distract from the overall meaning. Consider the following example from a proposed bank account management program:

```
accountNumber = verifyUser( username, password )
depositAmount = getDepositAmount()
newBalance = applyDeposit( accountNumber, depositAmount )
addMonthlyInterest( accountNumber )
```

You can get a high-level idea of what is going on, even without knowing the details of how each of those functions work.

- **Code is easier to test and debug** -- individual functions can be tested separately before they are combined in a larger program.

- **Code is easier to re-use** -- a well-written (general-purpose) function can be reused in many situations, such as len() and input().

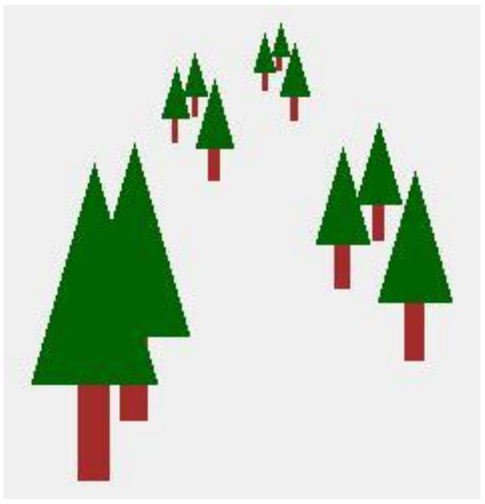
The following principles apply to the design and use of functions and methods:

- A function should perform only one task. You should be able to describe the purpose in a sentence. Notice I'm not saying **how** the function works, just what it does.
- Communication from some portion of code to a function is usually via the arguments/parameters and the return statement:
    - A function should get its inputs via parameters -- not from global variables, or the user. An exception to this would be a function whose defined task is for user input.
    - A function should generally return a value, not display to the user. An exception to this would be a function whose defined task is program output.
- In general, the user of a function is not concerned with how the function performs its task, but only with what it does and what information (parameters) it needs. For example, it is not necessary for you to know *how* math.sqrt() works, only what you must pass in to use it. (By the way, who is the user of a function? A programmer!)

Creating a program that is well designed (using programmer-defined functions) is a skill that comes with practice. As a new programmer, you should look at examples and pay attention to their design.

- What functions does the program use? How do those functions add to the design?
- In homework assignments, what functions are you being directed to write? Do you understand the reasoning behind writing that code as a function?

3 Functional Decomposition -- a Graphical Example



This week's homework assignment will give you some practice with functional decomposition by exposing you to some simple drawing tools and challenging you to compose a scene in a hierarchical fashion. This very visual illustration of composition/decomposition should help you develop an understanding of these concepts. Review Chapter 3 in the text in addition to reading this note.

3.1 Scene Composition

Look at the image above. What do you see? Was your first thought "I see triangles and rectangles"? Probably not, although that is certainly true. Even this crude image probably conveys the idea of trees to most viewers.

When one attempts to compose a scene like this, it makes sense to think in terms of "high-level" **scene elements** rather than the raw shapes. The scene at right could be produced by drawing 22 simple shapes: 11 rectangles and 11 triangles. However, this would be a tedious approach, and would not scale well to larger scenes. Instead, an approach that involves thinking in terms of larger elements is helpful. For example, one could think of this scene as being made up of 11 trees, where each tree is composed of one rectangle and one triangle. The trees may be of different sizes and locations in the scene, but the relative position and size of the two shapes to each other is the same in each tree. This means that we can "define" a tree to be made up these two shapes, and then we can "define" the scene to be made up of a series of trees.

If you look more closely at the scene, you may be able to see that it has even more structure than this. Ignoring the two largest trees for a moment, look at the rest of them. Can you see that there are really three "clusters" of trees? The clusters have different sizes and locations, but the relative size and location of each of the three trees is the same within each cluster. This means we can "define" a cluster as a larger scene element.

With all this in mind, we can now say that this scene is made up of just 5 elements: three tree clusters and two individual trees. That's a lot easier to understand than 11 rectangles and 11 triangles!

3.2 Using Functional Decomposition to Create the Scene

Some of the code used to create the tree scene we have been discussing is shown below. (Unimportant details have been left out. The entire program is available as a sample program on the class web site.) We will be looking at a total of 3 function definitions. Let's take a look at the 'main' function first:

```
def main():  # draw things on the canvas
    draw_tree_cluster(0, 0, 50)
    draw_tree_cluster(-40, -30, 65)
    draw_tree_cluster(60, -120 , 120)
    draw_simple_tree(-80, -150, 140)
    draw_simple_tree(-100, -180, 160)
```

Notice the main() function has just 5 statements -- one for each of the 5 high-level scene elements. In effect we are thinking about the problem of drawing the scene in an abstract way. All we need to do to create this scene is to draw 5 elements. This is a very easy way to visualize the task! We are breaking a big problem (drawing a scene that ultimately has 22 shapes) down into smaller problems (drawing a tree and drawing a cluster of trees). Notice that each time we call one of the functions, we provide information, the **arguments**, that customizes what the function does. Now let's solve each of the smaller problems...

Let's look at the 'draw_tree_cluster' function next:

```
# Draws a cluster of three trees. The parmeters x and y specify
# the location of a point at the center of the bottom edge
# of the tree trunk of the largest tree in the cluster.
# The last parameter is the "size" of the cluster -- the distance
# in pixels from the bottom to the top of the cluster.
def draw_tree_cluster(x, y, size):
    draw_simple_tree(x - size * 0.15, y + size * 0.5, size * 0.5)
    draw_simple_tree(x - size * 0.3, y + size * 0.3, size * 0.6)
    draw_simple_tree(x, y, size * 0.8)
```

At a high level of abstraction, we can say "to draw a tree cluster, simply draw three trees at specific locations and with specific sizes." So, the body of this function has just three statements. Once again, we are breaking a big problem down into smaller problems. Here are some key things to notice in the function definition above:

- The function has **parameters** that specify the location and size of the tree cluster. When the function is **called**, **the values of the arguments in the function call are passed into the**

parameters of the function definition. For the simple function calls in this sample program, the arguments must match the parameters in three ways: **the number, order and data types must be the same**.

- To ensure that the tree cluster will look correct when drawn in different locations and sizes, the location specified for each tree must be "relative to" the location specified for the cluster. In addition, every "distance" within the cluster (the sizes of the individual trees and the location offsets) must be proportional to the size of the cluster. The expression (x - size * 0.15) is an example of this: the 'x' coordinate of the location of the first tree drawn is offset from the cluster 'x' location by 15% of the cluster size.
- The order in which scene elements are drawn may matter. When two shapes overlap each other, the shape drawn later shows up "on top of" the shape drawn earlier; shapes drawn later may partially or fully obscure (cover up) shapes drawn earlier. In effect this means that a scene needs to be drawn "from back to front."

Up until this point, all of our code has been "abstract" -- we have not actually written code that draws a shape yet. We've been thinking about trees and clusters, but not about what it takes to create a tree. However, that's all that's left to complete our solution to the problem:

```
# Draws one tree. The parmeters base_x and base_y specify
# the location of a point at the center of the bottom edge
# of the tree trunk. The last parameter is the height of
# the tree. All parameters have units of pixels.
def draw_simple_tree(base_x, base_y, height):
    # draw trunk
    trunk_x1 = base_x - height * 0.05
    trunk_x2 = base_x + height * 0.05
    trunk_y1 = base_y
    trunk_y2 = base_y + height * 0.5
    canvas.rectangle([[trunk_x1, trunk_y1], [trunk_x2, trunk_y2]], \
        fill='brown', width = 0)
    # draw crown
    # the polygon has 3 points, peak, lower left (LL), and lower right (LR)
    LL_x = base_x - height * 0.2
    LR_x = base_x + height * 0.2
    L_y = base_y + height * 0.3
    canvas.polygon([[base_x, base_y + height], [LL_x, L_y], [LR_x, L_y]], \
        fill='darkgreen', width=0)
```

This function performs a series of calculations and draws two shapes (one rectangle and one triangle) to create a tree. Notice how much detail there is for each shape to be drawn. Fortunately, we need do this only for TWO shapes rather than 22. You can read more about the 'rectangle' and 'polygon' functions in the Gui Documentation reading.

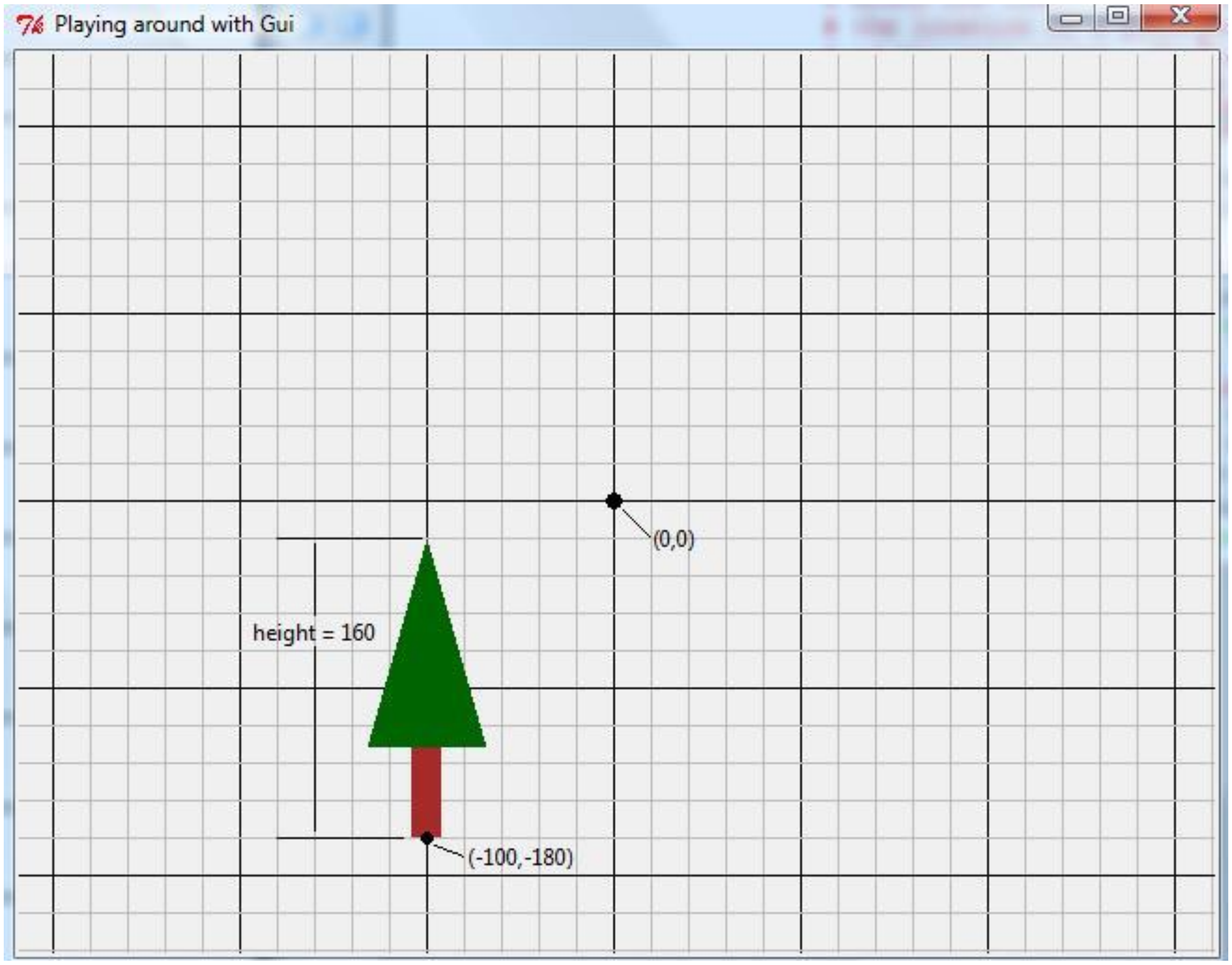
This program features a function "hierarchy" that matches a "top down" approach to problem solving. The entire scene was described abstractly as being made up of just 5 elements: three tree clusters and two trees, each with a specific location and size. A tree cluster, in turn, was described abstractly as being mad up of three trees, each with a specific location and height **relative to the location** and **proportional to the size** of the tree cluster. Finally, a tree was defined as being made up of two shapes with locations and sizes determined by the parameters specifying the tree. As a result of the functional decomposition process, the "hard part" of the program -- drawing a tree -- needs be done just one time. Once we have defined a **parameterized function** (i.e. a function with parameters) to draw a tree, we can simply call that function any time we want to draw a tree -- just like you call a built-in function any time you want to get user input or print something on the console!

4 Details of TreeTest.py

The sample program TreeTest.py demonstrates a structured approach to image composition by defining functions named 'draw_simple_tree' and 'draw_tree_cluster'. How does this code work? Let's deconstruct the effect of one high-level function call to see what is happening. Here is one statement from the 'main' function:

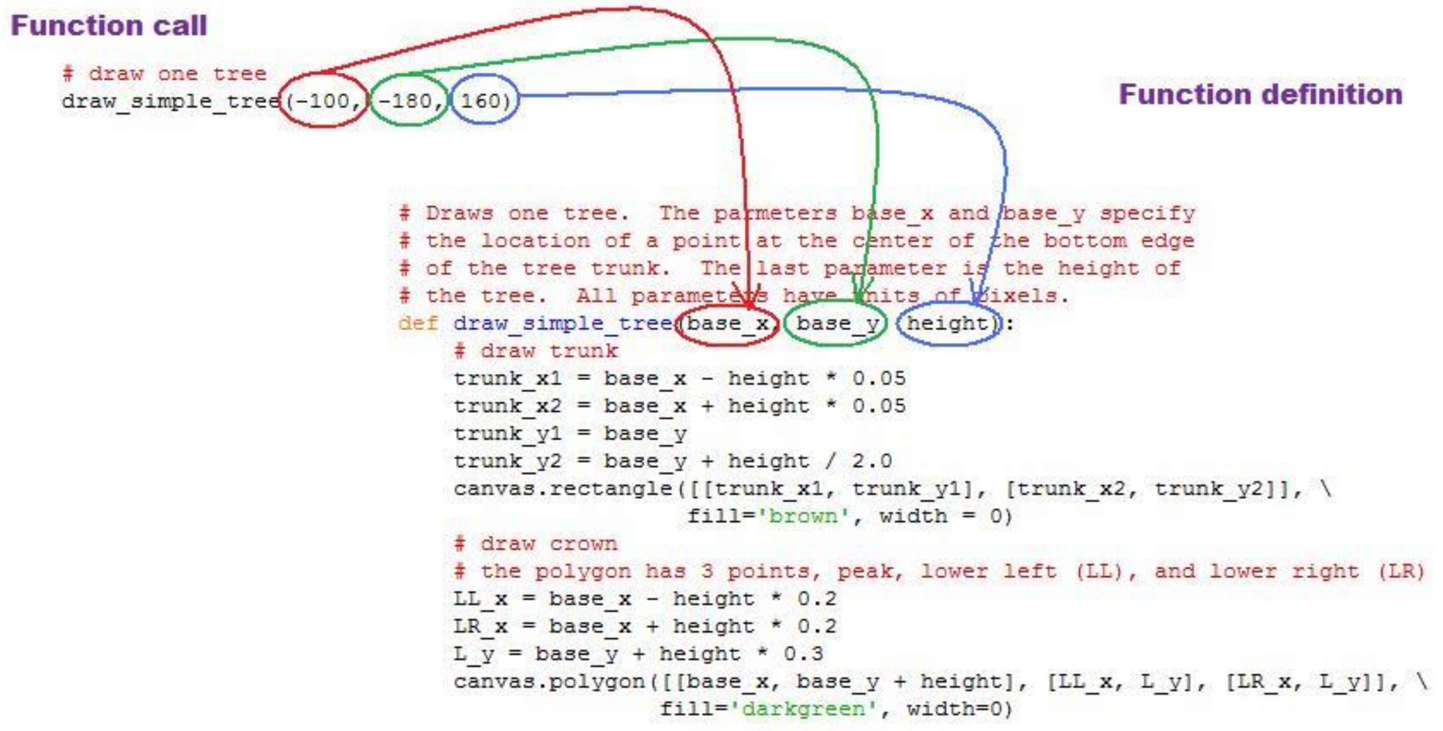
```
draw_simple_tree(-100, -180, 160)
```

This statement means "draw a tree planted at location x=-100, y=-180 with a height of 160 pixels." The "origin" of coordinates (0,0) is located in the center of the graphics window in our "Gui3" system, so this tree will be "planted" to the left of center and below the center line. Here is what that looks like (gridlines are shown every 20 pixels):



Notice that the center of the base of the tree is located at the point (-100,-180) on the grid. Notice also that the height of the tree is 160 pixels. The purpose of the 'show_simple_tree' function is to allow the programmer to use an "abstract" definition for a tree. This means that the programmer does not need to think about rectangles or polygons -- the actual shapes used to create the tree -- when drawing a scene. Instead, she can just think about planting trees! This is much more natural. Let's see how the code inside the 'draw_simple_tree' function definition accomplishes this.

The first thing to consider is what happens when a function is called. When a function is called, **the values of the <u>arguments</u> contained in the function <u>call</u> are passed into the <u>parameters</u> of the function <u>definition</u> in the order listed**. (Python's "keyword arguments" allow this order to be altered, but they are not used in this function call.) Here is a visual representation of this process for the statement above:



```
# draw one tree
draw_simple_tree(-100, -180, 160)
```

```
# Draws one tree.  The parmeters base_x and base_y specify
# the location of a point at the center of the bottom edge
# of the tree trunk.  The last parameter is the height of
# the tree.  All parameters have units of pixels.
def draw_simple_tree(base_x, base_y, height):
    # draw trunk
    trunk_x1 = base_x - height * 0.05
    trunk_x2 = base_x + height * 0.05
    trunk_y1 = base_y
    trunk_y2 = base_y + height / 2.0
    canvas.rectangle([[trunk_x1, trunk_y1], [trunk_x2, trunk_y2]], \
                     fill='brown', width = 0)
    # draw crown
    # the polygon has 3 points, peak, lower left (LL), and lower right (LR)
    LL_x = base_x - height * 0.2
    LR_x = base_x + height * 0.2
    L_y = base_y + height * 0.3
    canvas.polygon([[base_x, base_y + height], [LL_x, L_y], [LR_x, L_y]], \
                   fill='darkgreen', width=0)
```

After these values are passed into the parameters, the statements in the body of the function are executed. Notice that we can use the values of the parameters in formulas right away (even in the first statement in the function body). This is true because these parameters have already been assigned values. That's why every function call must contain whatever arguments are required by the function definition.

Drawing a simple tree like this is a two step process. First, a rectangle is drawn to represent the tree trunk. Then, after that, a triangle is drawn (using the 'polygon' function) to represent the "crown", or top, of the tree.

To draw the rectangle, we need to identify the locations of two diagonally opposite corners of the rectangle. These values are calculated and stored in the variables 'trunk_x1', 'trunk_y1', 'trunk_x2', and 'trunk_y2'. How do we know where these points should be located? Well, the location of each point depends on the location of the tree itself (as defined by the values of 'base_x' and 'base_y'), and also the height of the tree. Every tree drawn by this function has the same "shape" -- the same ratio of width to height. So, if we make the tree taller, it also gets proportionally wider. This means that every time we calculate a distance (e.g. an offset in the location of a point), that distance must be proportional to the height of the tree. The width of the tree trunk is defined to be 10% of the height of the tree, centered over the point (base_x, base_y). Here are the calculations for the two x coordinates of the tree trunk:

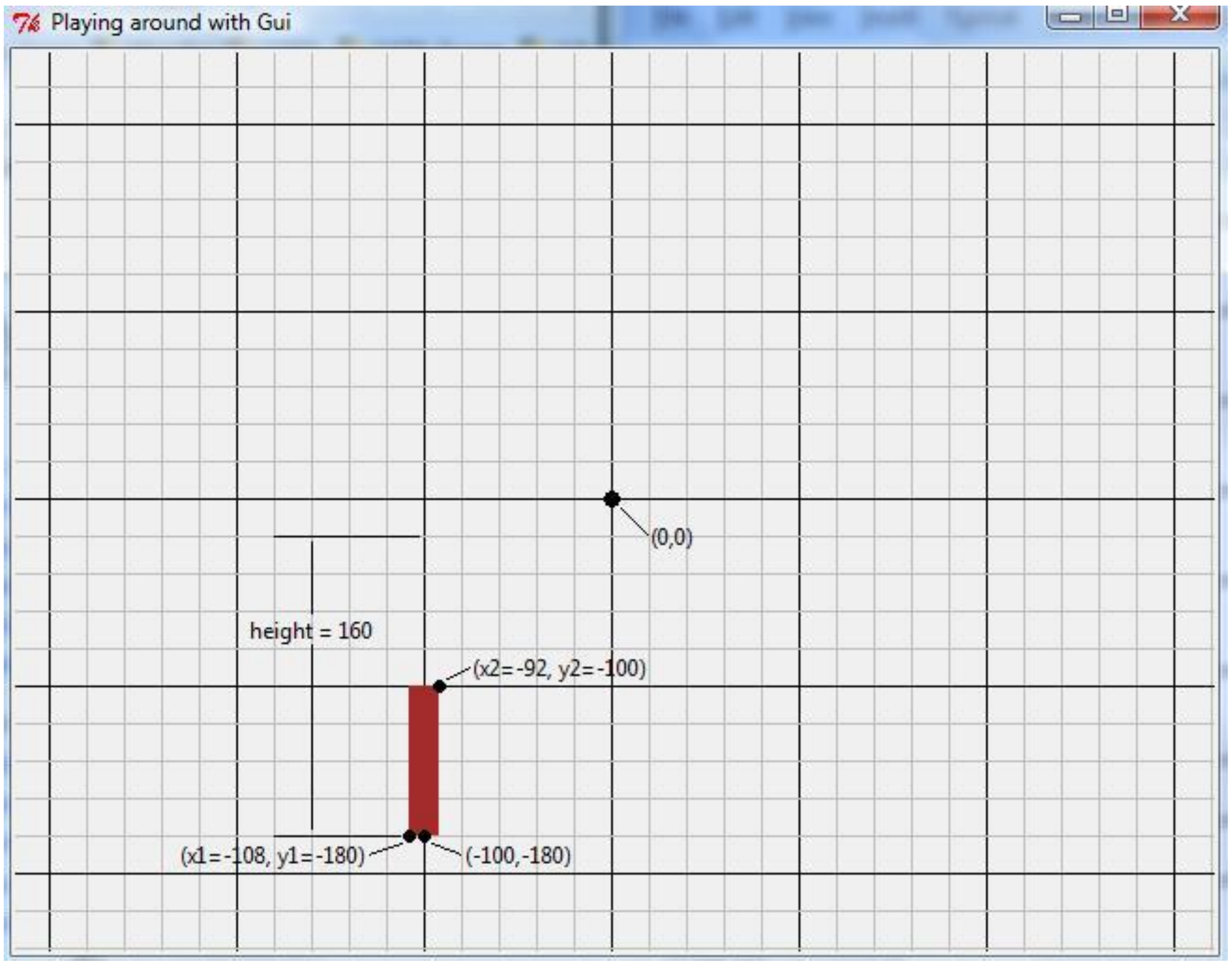
```
trunk_x1 = base_x - height * 0.05       # trunk_x1 = -100 - 160 * 0.05 = -108
trunk_x2 = base_x + height * 0.05       # trunk_x2 = -100 + 160 * 0.05 = -92
```

Notice that 'x1' is 5% of the height of the tree to the left of 'base_x' and 'x1' is 5% to the right of 'base_x'. That's how we center the rectangle over 'base_x'.

The height of the rectangle that makes up the tree trunk is set to half the tree height. Notice how trunk_y2 is calculated. To draw the tree trunk, a rectangle is drawn with a fill color of 'brown' and no border. Here is the statement that does that:

```
canvas.rectangle([[trunk_x1, trunk_y1], [trunk_x2, trunk_y2]], fill='brown', width = 0)
```

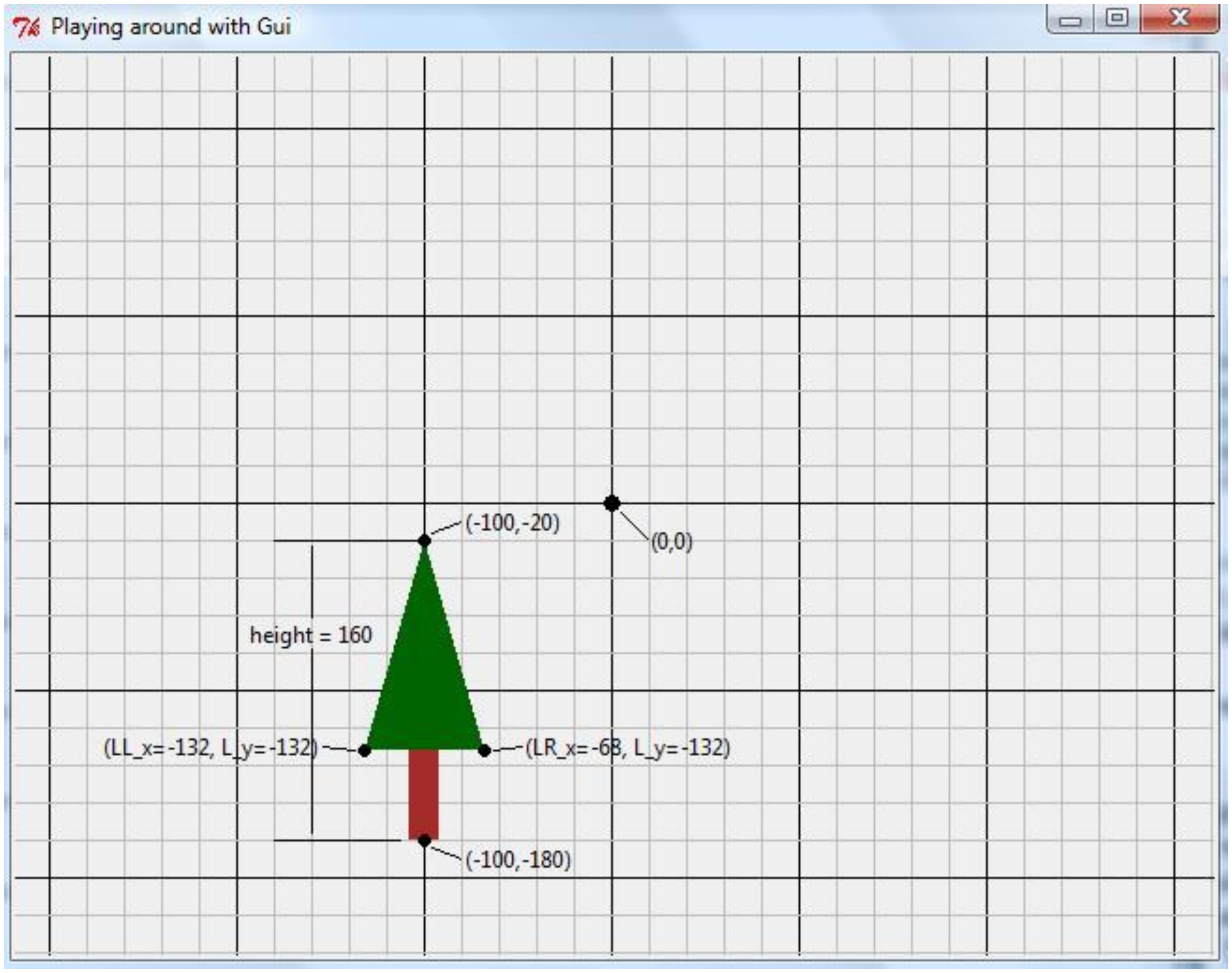
Here is the result after this rectangle is drawn:



Drawing a triangle to represent the "crown" of the tree is next. The 'polygon' function is used to draw the triangle. To use this function, we need calculate the coordinates of each of the three vertex points on the triangle. The top point is the easiest. This point is located 'height' pixels directly above the "reference location". So, its x-coordinate is located at 'base_x' and its y-coordinate is located at (base_y + height). The formulas for calculating the locations of the other points are shown in the code listing above. Once these values are calculated, this statement draws the triangle:

```
canvas.polygon([[base_x, base_y + height], [LL_x, L_y], [LR_x, L_y]], \
     fill='darkgreen', width=0)
```

Here is the result after the triangle is drawn:



Notice that the triangle partially covers up the rectangle. More recently drawn shapes can partially or fully cover up shapes drawn before them. Keep this in mind when you design a scene -- you can use it to artistic advantage.

You can think of the 'draw_simple_tree' function as having the effect of "translating" an abstract description of a tree ("draw a tree planted at location x=-100, y=-180 with a height of 160 pixels") into an actual drawing of a tree. In fact, one can say in general that functions create higher-level "abstractions" in a computer program. The 'draw_tree_cluster' function in the 'TreeTest3.py' sample program carries this idea a bit further by creating the abstraction of a scalable cluster of trees. Notice how these tools are used to create an entire forest scene when the sample program is executed.