CS 1410 Assignment 5: Minesweeper

In this assignment, you will utilize the pyGtk toolkit to create a simple minesweeper game. This assignment will allow you to further develop your object-oriented programming skills by having you develop several classes as well as be able to traverse an object heirarchy within pyGtk. If you do not recall how to play minesweeper, it may benefit you to spend a few minutes re-acquanting yourself with it.

The assignment is split into 2 parts. Each part has a specific pass-off requirement and a due date.

For the first part of the assignment, you will develop the GUI for the slider game using pyGtk. This GUI will be created within its own class. The pass off for this part of the assignment will be displaying the newly created GUI.

In part two, you will create a separate class and implement the logic portion of the slider game.

Getting started

You will need to have the following at the top of your code:

```
import pygtk
import gtk
```

You will need to import any other libraries that your code utilizes (e.g. `random`).

Please be advised that you are not going to be walked through every line of code that you will write. You will need to spend considerable effort and time looking at the pyGtk documentation found here.

In this assignment we are dealing extensively with what we will call object aggregation. Simply put, object aggregation is where you have an object that can contain one or more other objects (widgets). In this case, you will begin with a main window element (`main_win`). The `main_win` will contain a `VBox` (`main_vbox`). This `main_vbox` contains both a `MenuBar` (`menu_bar`), and a table (`game_table`). For each cell within the table you will attach a button (`btn`). A cell is defined as the intersection of a row and a column. Underneath a button you might have a bomb, a number (indicating how many bombs surround a particular cell), or nothing. As in the original minesweeper game, you will be able to place flags on the buttons which you believe have bombs underneath them.

To help you visualize the above paragraph, consider the following:

```
Window (main_win)
    gtk.VBox (main_vbox)
            gtk.MenuBar(menu_bar)
                    gtk.Menu (menu)
                            gtk.MenuItem (menu_item)
                                    gtk.MenuItem
                                    gtk.MenuItem
                                    gtk.MenuItem
            gtk.Table (table)
                    gtk.Button (btn)
                    gtk.Image (img)
                    gtk.Label
                    ...
```

Note that an image or label may be under a button or the button may have nothing underneath it.

The GUI: Part I

First, we need to recreate the visible interface of the project. This will be taken care of by a class which you should call MinesweeperGUI. This class will take care of creating the window, menu, table, buttons and the other visual elements of the project. You will need to obtain an image for a flag and an image for a bomb that you can use within the project. The images should be smaller jpegs. They need not be an exact dimension, we will size them later on.

At the top of your program, you should define several global variables that will make things easier as we progress.

```
GAME_SIZE = 10              # size of game in buttons (i.e. 10x10)
CELL_SIZE = 20              # how big each buttons should be (in pixels)
IMG_FLAG = 'flag.jpeg'      # image to display for flags
```

```
IMG_BOMB = 'bomb.jpeg'      # image to display for bombs
NUM_BOMBS = 10              # how many bombs should be on the screen
```

We may add more globals later.

You can test the class by creating an instance of it and invoking the `run` method when you are ready.

```
m = MinesweeperGUI()
m.run()
```

MinesweeperGUI class

Here are some details about the implementation of the `MinesweeperGUI` class:

constructor

The constructor does not need to receive anything besides `self`. It will handle the setup of the visual elements. It may help to code the constructor after you have figured out what the other methods will do.

You should create a list that will store all of your buttons.

Call the following methods in order (each method will be described in more detail later):

- `createWindow`
- `createMenu`
- `createGame`

Finally, you need to tell the window to display all widgets (i.e. `self.win.show_all()`)

createMenu

This method receives no parameters other than `self`.

It creates a `gtk.Menu` object (`self.menu`). Then it calls the helper method `createMenuItem()` to create each menu item and add them to `self.menu` (to be discussed in the following section).

It creates 3 menu items, “New Game”, “Solve”, and “Quit”. These menu items should activate the methods `restart_handler`, `solve_handler`, and `destroy_handler`, respectively. Each of these methods will be discussed in the writeup below.

It creates a menu item called `root_menu` that has the title “Game”. It sets the submenu for `root_menu` to `self.menu`.

Next it creates a `gtk.MenuBar`, and appends `root_menu` to the menu bar.

Finally it packs the menu bar at the start of of `main_vbox`.

createMenuItem

This method receives two parameters besides `self`. The first is the title to display in a menu item. The second is the name of a handler function to call when the menu item is activated.

This method is used as a helper method for `createMenu()`. It creates a `gtk.MenuItem` with the title, connects the `'activate'` signal to the handler function, and appends the newly created menu item onto `self.menu`.

getSmallImage

This method should receive `self` as well as the filename of the image. (i.e. `bomb.jpeg`)

This method will take the flag or bomb jpeg image that you have downloaded, scale them, and get them ready to display. The steps are as follows:

- Create an instance of an `gtk.Image`
- Create a new pixbuf (`gtk.gdk.pixbuf_new_from_file`)
- On that pixbuf (pb), call `pb.scale_simple(CELL_SIZE, CELL_SIZE, gtk.gdk.INTERP_BILINEAR)` to scale the image (store

the resultant value).
- Set the image instance to the new scaled pixbuf `set_from_pixbuf`.
- return the image

createGame

This method will create all the buttons and pack them into the table. The method should receive only `self`. Create a class datamember of the `gtk.Table`, stored as `self.table`.

Loop through each potential location in the table. For a 10×10 table you would loop through each of the row/column intersections in turn. At each of those 'cell' locations, you will want to create a button, size it (`set_size_request`), append the button to your list, connect the clicked handler of your button and finally, attach the button to your table (`table.attach`).

Note: The clicked handler of the button can be set by doing something like `b.connect('button_press_event', self.clicked_handler)`

You finally need to put your table on the main vbox by utilizing the `pack_start` method of the VBox class.

createWindow

This method receives no parameters.

In this method, create the main window, give it a title, connect the `delete_event` to the `delete_handler()` method, connect the `'destroy'` signal to the `destroy_handler()` method, set the window to not be resizable, create the `main_vbox` and add it to the window. The actual handlers will be discussed later on.

To connect an event handler to a widget, use something like this:

```
widget.connect('event_type', self.handler, data)
```

where `data` is any arbitrary value that should be passed to the handler. Note that you do not put parentheses on the name of the handler; you are not calling it.

When you set the default size remember to make it dependent on the size variables that you calculated earlier and should have stored as datamembers.

delete_handler

Each event handler should receive the exact same arguments:

```
def handler(self, widget, event, data=None):
```

This is the handler for the 'delete_event' on the window. It needs to return `False`.

destroy_handler

Each signal handler should receive the exact same arguments:

```
def handler(self, widget, data=None):
```

This is the handler for the 'destroy' signal on the window, and the 'activate' signal on the 'Quit' menu item. It needs to tidy up everything associated with gtk, when closing window. It does this by calling:

```
gtk.main_quit()
```

restart_handler

This is the handler for the 'activate' signal on the 'New Game' menu item. For now it is a stub for part 2. Have it print out 'restart'.

solve_handler

This is the handler for the 'activate' signal on the 'Solve' menu item. When you click it, it should hide all the buttons on the screen to let you see what is underneath them. Do so via the `clearFlagsAndHideButtons` method described subsequently.

clicked_handler

This follows the pattern of 'event' handlers described above.

We will do more with this in part 2, for now you should test to see if the left mouse button was clicked (`event.button == 1`) or the right mouse button (`event.button == 3`). When the user clicks the left mouse you should print out a message "Left clicked" and the button should be hidden, when they click the right mouse button, you should call `toggleFlag` (described subsequently) and be able to see if your flag appears/disappears on that particular button.

We will change how to handle the left button click event in part 2.

clearFlagsAndHideButtons

This method does just what it says and will only receive 'self' as a parameter. It will loop through each button, hide it. If that button has an image set it will hide that image via `i.set_visible(False)` assuming `i` is the image. (See the gtk documentation on buttons)

toggleFlag

This method should receive `self` and a widget (button).

This method handles placing/removing the flag on top of the button. It will first check to see if there is an image on the button, if not, it will set the image that it first retrieves from the getSmallImage method. If the button already had an image and that image was visible, it will need to hide it. If the button had an image and it was invisible, it will need to display it. As described above, this method should be invoked when the user issues a 'right-click' event on a button.

You should read the gtk documentation on `Button` (and perhaps `Image`) to see how to check if an image on the button.

If you do not have version 2.2 or higher of pygtk installed, you will not be able to use the `get_visible` function to test for visibility. Instead, you should use something like:

```
if widget.flags() & gtk.VISIBLE:
```

To test for visibility.

updateDisplay

This will be a stub method until we develop it in part II.

run

This method takes no parameters. It will start the main gtk loop:

```
gtk.main()
```

After you correctly build this method, you should be able to see a window on your screen with the correct title and size.

Wrapping Things Up in Part I

By this time, your gui should work, i.e., you should be able to right click on a particular button, a flag will be placed. If a flag is already displayed, it will be hidden. We need to make a final few changes to make things a little more convenient to work with and to aid in programming part 2.

First, extend the `gtk.Table` class by creating your own class called `myTable`. Your class should recieve the same parameters as a `gtk.Table` and immediately invoke the parent class constructor. We are going to also store a list of cells. Each cell is a row/column intersection within a table. This cell will be a list of tuples `(b, index)`, where `b` is a button and `index` is the location in the table where we stored it. We will store a list of images and labels that will be displayed on the table. This code

should help:

```
class myTable(gtk.Table):
   def __init__(self, xsize, ysize):
     gtk.Table.__init__(self, xsize, ysize, True)
     numcells = xsize * ysize
     self.xsize = xsize
     self.ysize = ysize
     self.cells = []
     self.images = []
     self.labels = []
```

You should change your `MinesweeperGUI` class to utilize `myTable` instead of `Table`.

Your newly created class should have the following methods: (please note that each method of the class should have the first parameter as 'self'. If nothing else is indicated, nothing else should be received!)

attachButton

This method should receive a button and a row and column where it is to be inserted. It will call the `gtk.Table.attach` method appropriately. You also should append that button (`b`) to your cells list.

```
index = row * GAME_SIZE + col
self.cells.append((b, index))
```

getRowColOfButton

For a particular button (method receives a button), return the row/column of where that button is located in the table. Note that in the cells list we have stored indices. You should be able to convert the index to a row/column location and return that row/col as a tuple.

attachImage

This method receives an Image.

This method does the same thing as attachButton, but instead of storing in cells list, store the image in your image list. (Does not need to be a tuple, just put it in your list)

attachLabel

This method receives a Label.

This method does the same thing as attachImage, but instead of storing in cells list, store the label in your labels list.(Does not need to be a tuple, just put it in your list)

clearImages

Loop through all the images in the list and remove them.

clearLabels

Loop through all the labels in the list and remove them.

Once you have updated your `MinesweeperGUI` class to utilize the newly created `myTable` class. You are prepared to pass it off. To pass off Part I of the assignment, you should demonstrate to the labbie that you have a window with buttons. You should demonstrate that when you left click a button disappers. When you right-click on a button a flag is displayed or removed if already displayed.

The Logic: Part II

Part 1 was exciting, but our GUI doesn't do too much. In this part we will build out the functionality of our game so that things happen and we can have a bit more fun.

There are 2 classes that we are going to build to facilitate this part of the game: Cell and MinesweeperLogic. The Cell class is a very minimal class that allows us to track what should be in each row/column intersection of our table. MinesweeperLogic takes care of gameplay, placing bombs, and counting how many bombs are near a particular cell.

Let us begin with the `Cell` class. Once again, expect each method to receive `self` as the first parameter. Think of a Cell as being the square that is UNDERNEATH a button within your table. If the cell is visible (the button on top has to be gone). If the cell is not visible it is because there is a button on top of it.

Cell Class

constructor

Doesn't take any arguments. There are 2 datamembers that each cell should keep track of: contents, visibility. Initialize these within the constructor. Visibility should be set to `False` (by default all cells are invisible OR there are always buttons on top of each cell). The contents can be initialized to an empty string.

setVisible

Sets visibility to `True`

getVisible

Duh!! Return the appropriate data member.

getContents

Duh again!!

setContents

Receives a value and stores that in respective data-member.

That's it for the `Cell` class! Ahhhhhhhhhh, don't we wish all classes were that easy!

MinesweeperLogic class

constructor

Should recieve the size of the board and store it (`self.size`). Then invokes the `reset()` method of the same class.

reset

We are going to utilize a list to keep track of where our bombs are at, and other information about our board. This list will be a class datamember, should be called `self.bomb_locations`. Initialize the empty list. Another point: Each list item will actually be another list. This is known as a 2-dimensional list, a nested list, or a list within a list. I.e. `a=[[1,2],[5,6],[8,9]]`. For now you can just create a list, but remember that when we add things to it we should add entire lists. You ought to perhaps create the sample `a` list above from within the python shell and make sure that you know how to create a 2-d list as well as how to access indexes within the list.

Now, for each row/column intersection of your table (should be able to determine from `self.size`) you need to create an instance of a Cell and put it in your bomb_locations list. Here is some code that will help set up your list correctly:

```
for row in range(self.size):
    r = []
    for col in range(self.size):
        c = Cell()
        r.append(c)
    self.bomb_locations.append(r)
```

Okay, you can copy/paste but please make sure that you understand what is happening, otherwise you will be stumped

later on. It may be a good idea to loop through each item in the list and display the contents of each cell (via the `getContents` method).

Finally, your `reset` method should call `setBombLocations` and `setBombCounters` respectively and which will be discussed later.

getBombLocations

Returns `self.bomb_locations`.

setBombLocations

This method should randomly choose a row and column of your board and set the contents of those cells to 'bomb'. If you haven't done so already, create a global variable called `BOMB` and set its' value to `'bomb'`. Utilize the global variable when setting the contents of the cell. Also, if you don't have a global variable called `NUM_BOMBS`, please create one and initialize it to 10 (arguably this number should be based on the size of the board, so keep that in mind if you choose to do the optional part of the assignment at the end). Anyways, you need to create `NUM_BOMB` bombs in your list in random locations.

setBombCounters

This method takes care of figuring out how many bombs are in adjoining cells and storing that amount in the cell contents. For each row/column intersection in your table, check to see if the contents of that particular cell is NOT a `BOMB`. If it is NOT a bomb, set the contents of the cell to whatever is returned by `setBombCountNumbers` (discussed in the next section).

Here is some code that may help:

```
def setBombCounters(self):
    for row in range(self.size):
        for col in range(self.size):
            if self.bomb_locations[row][col].getContents() != BOMB:
                self.bomb_locations[row][col].setContents(self.setBombCountNumbers(row,col))
```

setBombCountNumbers

This method should receive a row and column value and examine the adjoining cells to see if there is a bomb in them. Remember that you should look each possible direction (there should be 8). For each direction, you need to invoke a call to `isBomb` to see if it is a bomb or not at that particular location. Note that `isBomb` returns a number, you need to add this in to a total. If I were looking at the cell directly above I would use the following test:

```
count += self.isBomb(row-1, col)
```

You would have a similar call for each of the 8 possible directions. The method should return the count of bombs that it found.

isBomb

Recieves the row and column that it should look at. If these are outside the bounds of the board, the method should immediately return 0. Otherwise the method will check the contents of the cell at that particular row/column (by looking at the cell within self.bomb_locations). If the contents at that row/column is a BOMB you will return 1, otherwise 0.

At this point it is *strongly* suggested to create an instance of your MinesweeperLogic class and print out the contents of each cell to see if it is appropriately setting the number of bombs as defined by NUM_BOMBS in random locations, and putting integers in other cell contents. If it is not, you need to correct methods above before proceeding.

Integrating the 2 classes

At this point, the GUI kind of works and the Logic works a little. We now need to integrate the two.

First we will create an instance of our MinesweeperLogic within the GUI class. You will need to do so within the constructor before the call to `createGame`.

Now add a call to the method `placeImagesAndLabels` at the end of your `createGame` method. Now let's figure out what that

new method is!

placeImagesAndLabels

This method resides in the GUI class, but relies on the Logic portion of our game to figure out where to place the bomb images and the numeric labels for bomb count. First, the method retrieves the bomb locations from the logic module via the `getBombLocations` method. The method then loops through each possible row/column combination and checks to see what the contents of the corresponding bomb location cell list is (`getContents`). If it is a BOMB, you will need to call `self.getSmallImage` (make sure you pass in IMG_BOMB so it gives you a scaled bomb image) and then attach that image to the table via a method that you created within the `myTable` class (`attachImage`). Otherwise (the cell contents is not a bomb) if the value of centents is greater than 0, you would need to create a gtk.Label, set the text of that label to whatever the cell contents is, and attach the label to the table.

updateDisplay

You should have developed a stub for this in Part 1, now we will edit it to do the following:

- get the bomb locations from the logic module
- loop through each table row/column intersection to see if that particular `Cell` instance is visible(`getVisible`), if so, you should hide the corresponding button at that particular row/column.

processCells

This method should reside in your GUI class. It will be called when the user clicks the left mouse button (put the appropriate code to invoke this method in the clicked event handler location). It will receive `self`, and button as its' only arguments and do the following:

- Find the row/column of that particular button by calling `getRowColOfButton` of your table class.
- retrieve the bomb locations list
- if that particular row/column is a BOMB (the game is now over) (check it via the `getContents` method)
    - Call `clearFlagsAndHideButtons` so that they can see the entire board and all the bombs.
    - print 'You landed on a bomb, Game over!'
- otherwise
    - call `checkCell` from the Logic module

restart_handler

This was a stub in part 1, now it should do the following:

- Call the logic `reset` method
- Clear all the table images (`self.myTable.clearImages`)
- Clear all the table labels (`self.myTable.clearLabels`)
- Clear all the flags and hide buttons (`clearFlagsAndHideButtons`)
- Replace all the images and labels (`self.placeImagesAndLabels`)
- Update the display
- Execute the `show_all` method of the main window (to reshow all the graphical elements)

checkCell

This method resides within the Logic module. Other than `self`, it receives a row and column as a parameter. This method takes care of hiding other contiguous cells as required in Minesweeper. After a particular cell is clicked we need to check to see what else we can hide. This method calls itself recursively, so you must provide a base case (a stopping point wherein the method will *stop* calling itself).

- First, if the row and column are outside the game bounds we can just return without any additional processing.
- If that particular location is already visible (check that cell via the `getVisible` method) we can return without any

additional processing.

- If the contents of that particular cell is a number (make sure that it is >0), we can set that particular cell visible.
- Otherwise, there must be nothing in the cell contents, so go ahead and expose the cell AND then call check cell for the cell above, right, below, and left of the current cell we are looking at.

Here is most of the code for you:

```
def checkCell(self, row, col):

    #outside bounds
    if row < 0 or row >= self.size or col < 0 or col >= self.size:
        return

    if self.bomb_locations[row][col].getVisible()==True:
        return

    cell_contents = self.bomb_locations[row][col].getContents()
    #if it is a number, unhide that cell
    if cell_contents > 0 and cell_contents != BOMB:
        self.bomb_locations[row][col].setVisible()

    else:
        self.bomb_locations[row][col].setVisible()

        #call `checkCell` on the cell to the right
        #call `checkCell` on the cell to the left
        #call `checkCell` on the cell above
        #call `checkCell` on the cell below
```

Finishing up Part 2

- Make sure that your data handler for the left mouse button click event only has a call to `processCells` and then `updateDisplay`.
- Your solve handler should hide all buttons/flags on the board and then call `updateDisplay`

That should be it. You should demonstrate the functionality of the game in order to pass it off. In particular, the following:

- Your game should appropriately restart, solve, quit.
    - when restarting, your game board should not be static. You should have updated bomb locations and numbers in each cell.
- Your game should end when you click on a bomb.
- When you click on a non-bomb cell, it should be hidden as well as relevant contiguous cells.
- When you right-click a button, a flag should be displayed/removed as appropriate.

Optional

You should only proceed to this section if you have correctly implemented *ALL* of the above. The game is lacking in a few key areas, you could do any of the following:

- Check for a win
    - If you have appropriately marked all the bombs with a flag
    - if all the cells are already hidden, it should automatically decide that the game has been won
    - if the user clicks on something (i.e. the windows version is a smiley face) it will check to see if you have placed flags appropriately to determine the win.
- Change the complexity of the game through the use of a menu option:
    - easy (existing 10x10 grid with 10 bombs)

    - med (larger grid, more bombs)
    - difficult (largest grid, more bombs)
- Create a timer somewhere on the board that counts how many seconds you took to solve the game. Should update each second.
- Prettify the game (if you do this option, it will have to ROCK pretty hard to qualify for any bonus points)
    - colors
    - menu titles (fonts)
    - customized flags and bombs