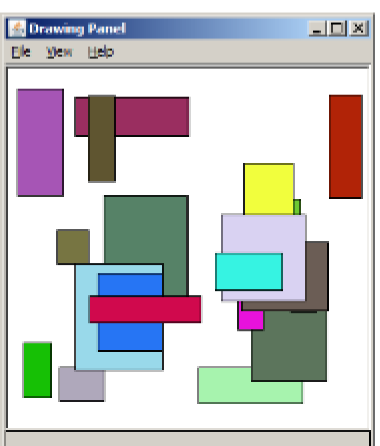
CSc 110, Autumn 2016
Programming Assignment #8: Tiles (20 points)
Due Thursday, October 25, 2016, 11:30 PM

thanks to Mike Clancy of UC Berkeley and Marty Stepp of Stanford

Part 1:

This project will give you insights into how operating systems manage multiple programs' windows. Its implementation focuses on using lists. Turn in a file named `tile_manager.py` online using the turnin link on the Homework section of the course web site. You will need `drawingpanel.py` from the Homework section of the course web site; place it in the same folder as your program.

Program Description:

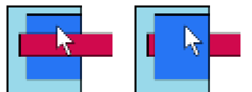


In this assignment you will write the logic for a graphical program that allows the user to click on rectangular tiles.

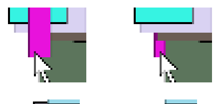
You should store all of the information needed to draw each tile in an appropriate structure and store one of these structures for each tile in a list. The order tiles are stored in the list should determine their drawing order. For example, consider the tall tile that overlaps the wide tile in the upper left corner of the screenshot at right. The two tiles occupy some of the same (x, y) pixels in the window but the wide one was drawn first because it occurred before the tall one in the list. When the tall tile was drawn later, it covered part of the wide tile. The list's ordering is called the 3-dimensional ordering or *z-ordering*.

The provided code reacts to mouse and key presses. You must write the code to create and display the tiles and to change what is displayed when a key or mouse button is pressed. Depending on the kind of input, different actions occurs:

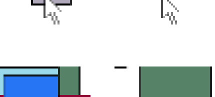
- If the user clicks the **left mouse button** while the mouse cursor points at a tile, that tile is moved to the very *top* of the z-ordering (the end of the tile list).



- If the user clicks the **left mouse button** and holds down the **Shift key** while the mouse cursor points at a tile, that tile is moved to the very *bottom* of the z-ordering (the start of the tile list).



- If the user clicks the **right mouse button** while the mouse cursor is pointing at a tile, that tile is removed from the tile list and disappears from the screen.



- If the user clicks the **right mouse button** and holds the **Shift key**, *all tiles that occupy that pixel* are removed from the tile list and disappear from the screen.

- If the user types the **N key** on the keyboard, a new randomly positioned tile is created and added to the screen.
- If the user types the **S key** on the keyboard, the tiles' order and location are randomly rearranged (*shuffled*).

If you use a Mac with a 1-button mouse, you can simulate a right-click with a Ctrl-click (or a two-finger tap on the touch pad on a Mac laptop).

If there is no tile where the user clicks, nothing happens. If the user clicks a pixel that is occupied by more than one tile, the top-most of these tiles is used. (Except if the user did a Shift-right-click, in which case it deletes all tiles touching that pixel, not just the top one.)

Note that **the code to detect mouse clicks and key presses is provided for you in the skeleton code**.

Implementation Details:

Your `tile_manager` program should store a list of tiles and a `DrawingPanel` as global variables. The various functions listed below will cause changes to the contents of that list and `DrawingPanel`.

The following sections describe in detail functions you must implement in your `tile_manager` class. The functions listed below are called by the graphical user interface ("GUI") in response to various mouse clicks, passing you an event object from which you can get the x, y coordinates where the user clicked by accessing `event.x` and `event.y`. If the coordinates passed do not touch any tiles, no action or error should occur. After any click or press, re-draw all of the tiles in your list.

`def **raises**(event)`
Called when the user left-clicks. The event containing the coordinates where the user clicked is passed in. If these coordinates touch any tiles, you should move the topmost of these tiles to the very top (end) of the list.

`def **lower**(event)`
Called when the user Shift-left-clicks. The event containing the coordinates where the user clicked is passed in. If these coordinates touch any tiles, you should move the topmost of these tiles to the very bottom (beginning) of the list.

`def **delete**(event)`
Called when the user right-clicks. The event containing the coordinates where the user clicked is passed in. If these coordinates touch any tiles, you should delete the topmost of these tiles from the list.

`def **delete_all**(event)`
Called when the user Shift-right-clicks. The event containing the coordinates where the user clicked is passed in. If these coordinates touch any tiles, you should delete *all* such tiles from the list.

`def **add_tile**(event)`
In this function you should add a tile at a random location (but fully on the screen as described in the **`shuffle`** description) to the end of your list of tiles.

`def **shuffle**(event)`
Called when the user types s. This function should move every tile on the screen to a new random x/y pixel position. The random position should be such that the square's top-left x/y position is non-negative and also such that every pixel of the tile is within the width and height provided class constants. For example, if the WIDTH is 300 and the HEIGHT is 200, a tile of size 20x20 must be moved to a random position such that its top-left x/y position is between (0, 0) and (280, 180).

Besides reacting to mouse and keyboard input, your program must also do the following:

- Add 50 rectangles of random sizes between 15 and 40 pixels wide and tall located at random locations but fully visible (as described in the **`shuffle`** description) to your list. They should be random colors. You can get a random color by calling `random_color()` on your `DrawingPanel`. For example:
```
p = DrawingPanel(100, 100)
color = p.random_color()
```
- Draw all of these rectangles on the `DrawingPanel` from bottom (start) to top (end) of your list.

The tiles should be added and drawn when the program starts. They should be redrawn whenever the contents of the list changes.

Development Strategy and Hints:

One of the most important techniques for professional developers is to write code in stages ("**iterative enhancement**" or "**stepwise refinement**") rather than trying to do it all at once. This includes testing for correctness at each stage before moving to the next one. The next few paragraphs contain a detailed development plan. Study it carefully and think about why we suggest the plan we do.

Start by writing empty "**stub**" versions of all the required functions so that the `tile_manager` class will run without errors.

We suggest that you write your `add_tile` first, then code to draw the tiles. You can then run the program to make sure that you can see the tiles appear on the screen. Add the required starting 50 rectangles to your list next. Then, add click-related functions one at a time and test each one individually to be sure it works before moving on to the next.

One part of this program involves figuring out which tile(s), if any, touch a given x/y pixel. You can figure this out by comparing the x/y position of the click to the x/y area covered by the tile. For example, if a tile has a top-left corner of (x=20, y=10), a width of 50, and a height of 15, it touches all of the pixels from (20, 10) through (69, 24) inclusive. Such a tile contains the point (32, 17) because 32 is between 20 and 69 and 17 is between 10 and 24.

If you have bugs or errors in your code, there are several things you can try. We recommend you print out the state of your list with temporary **`print` statements**. You should remove any such `print` statements before you turn in the assignment.

Style Guidelines and Grading:

As always, a major focus of our style grading is **redundancy**. As much as possible, avoid redundancy and repeated logic in your code. One powerful way to avoid redundancy is to create "helper" function(s) to capture repeated code. It is just

fine to have additional functions in your `tile_manager` beyond those specified here. For example, you may find that multiple functions in your program do similar things. If so, you should create helper function(s) to capture the common code.

You should not use any other data structures besides the list of tiles. You should use the list and its functions appropriately, and take advantage of its ability to "shift" elements as they are added or removed. Your list should not store any invalid or `None` elements as a result of any mouse click activity.

You should introduce **constants** for any hardcoded values that appear in the code.

You should follow good general Python style guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as functions, loops, and factoring common code out of `if/else` statements; good variable and function names; and not having any lines of code longer than 100 characters in length.

You should **comment** your code with a heading at the top of your class with your name, section, and a description of the overall program. Also place a comment heading on top of each function, and a comment on any complex sections of your code. Comment headings should use descriptive complete sentences and should be written *in your own words*, explaining each function's behavior, parameters, return values, and assumptions made by your code, as appropriate.

Your solution should use only material taught in class.

Part 2:

This part of the assignment will give you an opportunity to improve your testing skills. Turn in two files `whitebox.py` and `blackbox.py`.

Black Box Testing

Given the following function description, write a series of **up to 6 tests** as described in class, that test all important cases. DO NOT write the code to solve this problem!

Description

Write a function named `remove_consecutive_duplicates` that accepts as a list of integers, and modifies it by removing any consecutive duplicates. For example, if a list named `list` stores `[1, 2, 2, 3, 2, 2, 3]`, the call of `remove_consecutive_duplicates(list)` should modify it to store `[1, 2, 3, 2, 3]`.

White Box Testing

Given the following code, write up to 6 tests as described in class, that test all important cases. The code should find the longest sorted (increasing) sequence between the start and stop (inclusive) and return its length. The code may not be correct.

Code

```
def longest_sorted_sequence(list, start, stop):
    if (len(list) == 0):
        return 0
    max = 1
    count = 1
    for i in range(start, stop):
        if (list[i] >= list[i - 1]):
            count += 1
        else:
            count = 1
        if (count > max):
            max = count
    return max
```

Remember, each test should test one thing. It should have a descriptive error message. Each test should be able to discover a different type of error. Your testing code will be held to the same style standards as the rest of the code you have turned in this semester.