Nicholas Bowman, Sonja Johnson-Yu, Kylie Jue **Section #5**
CS106AP **July 24, 2019**

Section Handout #5: Graphics

1. Grid Drawing

Python is commonly used in industry and academia as a tool for managing and visualizing large amounts of data. Almost any visualization requires the use of some sort of grid on which you can orient your data. Python has several toolboxes for producing such grids automatically, and in this problem we'll build an understanding of how they work! We'll be creating a grid manually using the **`tkinter`** library. We'll primarily employ the following functions, which we reproduce here with the assumption that a canvas has already been created for us:

```
# Draw a line between (x1, y1) and (x2, y2), touching both points
# optional: fill='red' to set color ('black' is default)
# optional: width=2 to draw a thicker line (1 is default)
canvas.create_line(x1, y1, x2, y2)
canvas.create_line(x1, y1, x2, y2, fill='red')
canvas.create_line(x1, y1, x2, y2, width=2)

# Draws text on the canvas.
# anchor=tkinter.SW means the given x,y is SouthWest corner of the beginning
# of the string. Other common choices are W (West) and NW (NorthWest)
# optional: fill='red' to set color ('black' is default)
canvas.create_text(x, y, text=some_str_value, anchor=tkinter.SW)
canvas.create_text(x, y, text=some_str_value, anchor=tkinter.SW, fill='red')
```

The starter code for the problem comes with a **`main()`** function that's already outfitted to facilitate two behaviors:

- If your program is called with one integer command line argument (i.e. **`python3 draw-grid.py 10`**), it will produce a 500 pixel x 300 pixel grid, separated into that many rows and columns.
- If your program is called with two integer command line arguments, (i.e. **`python3 draw-grid.py 1200 800 25`**), it will use the first two arguments as the dimensions of the grid and the third as the number of rows and columns.

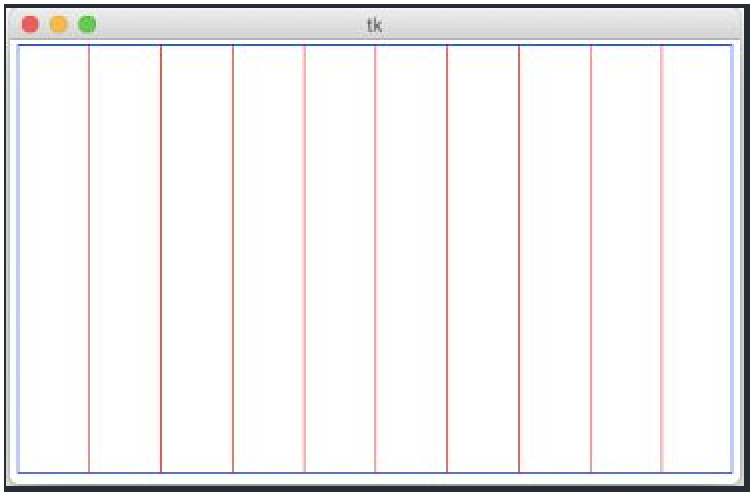
Your job is to complete the following function, which the **`main()`** function will then call:

```
def draw_grid(width, height, n):
    """
    Divides a canvas into n rows and columns
    """
    # make canvas of specified dimensions
    canvas = make_canvas(width, height)

    # TODO: your code here

    # required line to put window on screen
    tkinter.mainloop()
```

The function begins by making a canvas (delineated by a blue border) for you to separate into a grid using the **`make_canvas()`** function (which we provide) and ends by calling **`tkinter.mainloop()`** to put the window on the screen. It is your responsibility to actually draw the grid.

Milestone 1: Drawing column separators

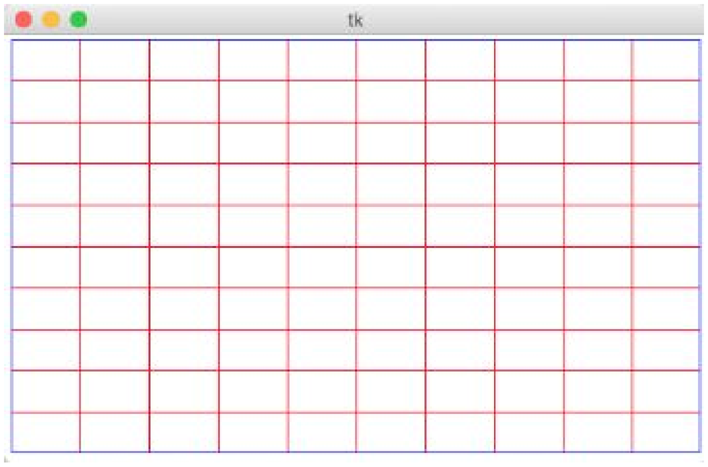
We'll begin by first dividing the canvas into **`n`** vertical columns, which is accomplished by drawing **`n - 1`** vertical red lines between said columns, like so:



To determine the x-coordinate of each line, compute its position as a fraction of the entire width of the canvas, then multiply that fraction by the canvas width. This fraction will by definition be a number between 0 and 1 and thus will need to be represented by the **`float`** data type. In your own code, this likely means that you'll simply use the `/` operator rather than the integer division `//` operator which you might be more used to. Each vertical line should extend the full height of the canvas.

Milestone 2: Drawing row separators

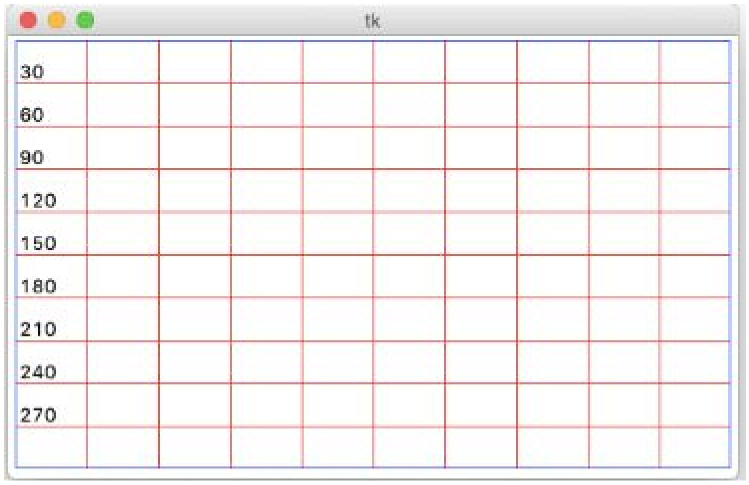
Now, you'll do the same thing to draw separator lines for the rows of the grid to produce a window like this:



Each horizontal line should extend the full width of the canvas.

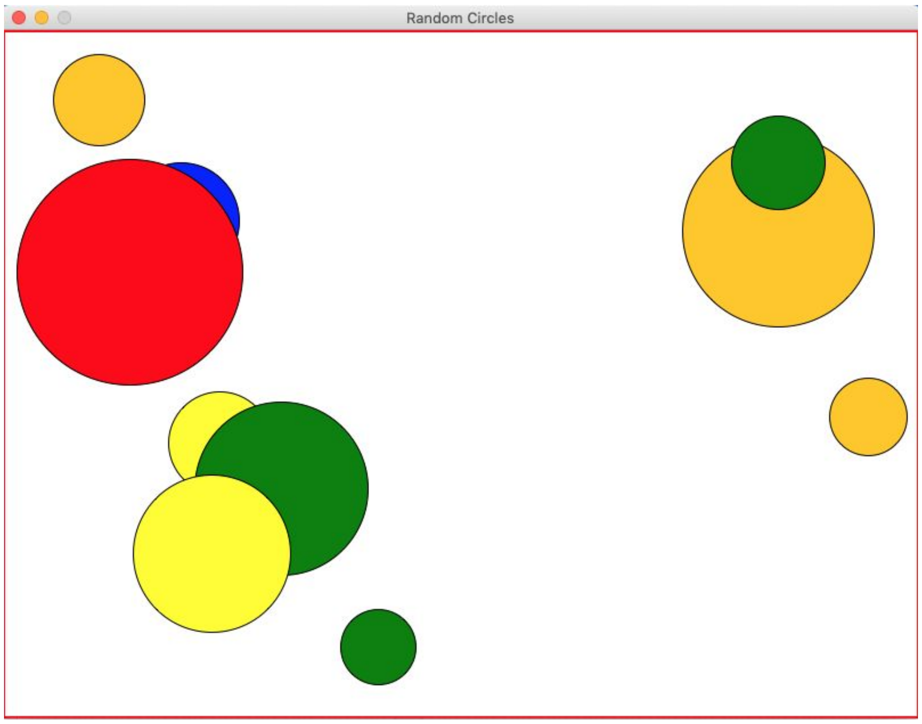
Milestone 3: Drawing labels

Finally, you'll label each of the row separation lines with its y-coordinate. Each label should be above and to the right of the leftmost end of the line, like so:



In **`tkinter`** nomenclature, we say that the anchor point for the label is **`anchor=tkinter.SW`**, which means that the coordinates we specify for the label are those for its SouthWest corner.

2. Random Circles



Write a program that draws a random number of circles, each of which has a random color, a random size, and a random position on the canvas. Be careful to make sure that none of the drawn circles are cut off by the edge of your canvas. You are provided with the constants **`WIDTH`** and **`HEIGHT`** (the canvas width and height, respectively), **`RADIUS_MAX`** and **`RADIUS_MIN`** (the maximum/minimum radius that each random circle may have), **`COLORS`** (a list of all possible circle colors), and **`N_CIRCLES_MAX`** (the maximum number of circles that may be generated in one run of our program. Note that each run should generate between 1 and **`N_CIRCLES_MAX`** circles inclusive on both ends). Specifically, your job is to implement the following function:

```
def make_all_circles(window)
```

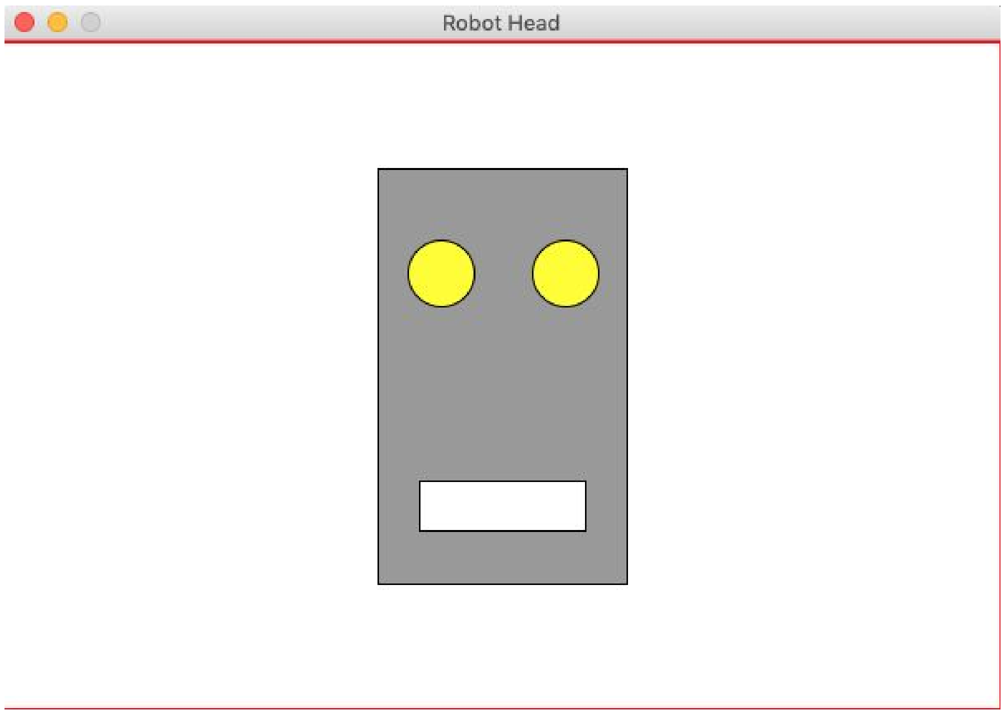
into which is passed a window and whose job is to do the random drawing of the circles. You might find the following functions helpful, some of which are found in the **`random`** module:

```
# creates an oval with a bounding box that has specified height and width
# the top left corner of the bounding box is at (x,y)
oval = GOval(width=width, height=height, x=x, y=y)

# returns a random integer between lower and upper, inclusive of both
# bounds
random.randint(lower, upper)

# returns a random number between 0 and 1
```

```
random.random()
```

3. Robot Face

Your job is to write a program that will draw a robot-looking face like the one shown in the following sample run:



This simple face consists of four parts—a head, two eyes, and a mouth—which are arranged as follows:

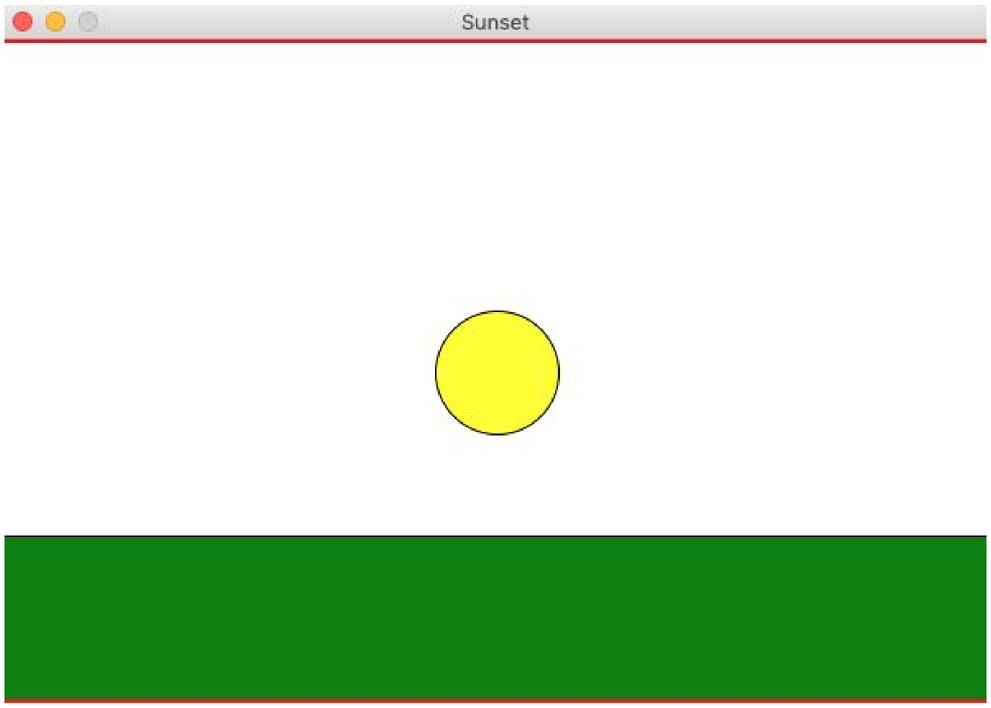
- The head. The head is a big rectangle whose dimensions are given by the named constants **`HEAD_WIDTH`** and **`HEAD_HEIGHT`**. The head is gray.
- The eyes. The eyes should be circles whose radius in pixels is given by the named constant **`EYE_RADIUS`**. The centers of the eyes should be set horizontally a quarter of the width of the head in from either edge, and one quarter of the distance down from the top of the head. The eyes are yellow.
- The mouth. The mouth should be centered with respect to the head in the x-dimension and one quarter of the distance up from the bottom of the head in the y-dimension. The dimensions of the mouth are given by the named constants **`MOUTH_WIDTH`** and **`MOUTH_HEIGHT`**. The mouth is white.

Finally, the robot face should be centered in the graphics window.

This is a long and intricate graphical function with many reasonable solutions. Before you work through it, think about what a sensible strategy would be to break it up into more manageable components. What sort of work is repeated with only superficial differences?

Additionally, consider how a program like this could be designed such that it's easy to modify. A large part of the process of programming is designing for the future; that is, ensuring that your programs are easy to extend to new situations. For example, how could you add ears or a body to your robot?

4. Sunset



Write a program that simulates a sunset. Your program should start off by creating a window, and drawing the sun centered in the window over a green horizon, as shown above. Your program should then animate the sun sinking beneath the horizon. If you'd like, you could change the color of the sun, the sky, or the horizon as the sun sets.

Your program is provided with the following constants:

```
WIDTH = 600
HEIGHT = 400

SUN_DIAMETER = 75
HORIZON_HEIGHT = 100
SUNSET_VELOCITY = 1.0;
PAUSE_TIME = 40 # MS pause b/w frames
```

5. Bonus: Pynary Bomb

A rogue Cal student has somehow gained access to a computer on Stanford premises and as a misguided attempt to demonstrate their technical superiority, has left an ill-intentioned program -- a 'Pynary bomb' -- for us to deal with. Left unchecked, there's no telling what this program might do: perhaps it'll delete the 106AP website and all your hard work on the assignments, or perhaps it'll constantly post low-quality content to SMFET (Stanford Memes for Edgy Trees, the ever-popular Facebook group), diluting Stanford's cultural credibility.

Fortunately for us, that student underestimated both your tenacity and skill with Python, and it's up to you to save the day. Your mission, should you choose to accept it, is to trace through the student's program (reproduced below) and to figure out the set of command line arguments that 'defuse' the bomb.

We've collected some intel about the program, which we're sharing with you here:

- The program is contained in the **`pynary_bomb.py`** file in the starter code.
- The author of this program is ill intentioned, without question, but is not duplicitous. The functions they've written are at least good-faith efforts in implementing the behaviour their names describe, although they might not always succeed.
- The bomb will not resist any attempts by you to make its workings more transparent, for example by printing the values of various variables throughout the program's execution.

```
def swap(li, idx0, idx1):
    temp = li[idx0]
    li[idx0] = li[idx1]
    li[idx1] = temp
    return idx1 - 1

def extends(d, k1, k2):
    copy = d[k1]
    copy.extend(d[k2])

def slicer(lst, val):
    lst = lst[val:]

def even_odd_counter(d):
    d_even_odd = {}
    for k in d.keys():
        for v in d[k]:
            x = v % 2
            if x not in d_even_odd:
                d_even_odd[x] = 0
            d_even_odd[x] += 1
    return d_even_odd

def foo(inp1, inp2, inp3):
    y = {'a': [1], 'b': [2,3], 'c': [4,5,6], 'd':[7,8,9,10],
'e':[11,12,13,14,15]}
    x = sorted(list(y.keys()), reverse=True) # sorts y.keys() in descending
order
    idx0 = 0
    while idx0 <= inp1:
        inp1 = swap(x, 0, inp1)
    if y[x[2]] != [2,3] and x[4] != 'a':
        print("BOOM!!")
        return
    slicer(x, 2)
    extends(y, x[0], x[inp2])
    if len(y[x[0]]) != 5:
        print("BOOM!!")
        return
    d_count = even_odd_counter(y)
    if d_count[1] != inp3:
        print("BOOM!!")
        return
    print("Phew! That was a close one.")
    print("You've defeated the Pynary Bomb, congratulations!")

def main():
    args = sys.argv[1:]
    print("Good luck...")
    foo(int(args[0]), int(args[1]), int(args[2]))

if __name__ == "__main__":
    main()
```