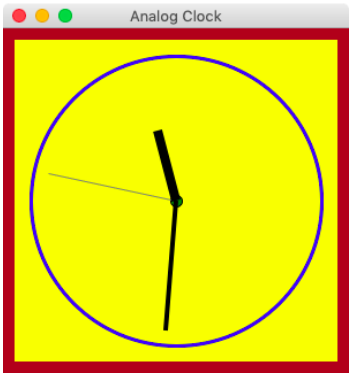
Lab 5: Analog Clock

01219116/117 Computer Programming II

In this lab assignment, we will create an analog clock application using Tkinter's canvas. Our application will look similar to the screenshot below.



Here is the detailed specification:

- The dimension of the application's window is initially 300x300 pixels.
- The application window is displayed with the title **Analog Clock**.
- The clock is updated every second.
- When the application window is resized, the clock display gets resized accordingly.

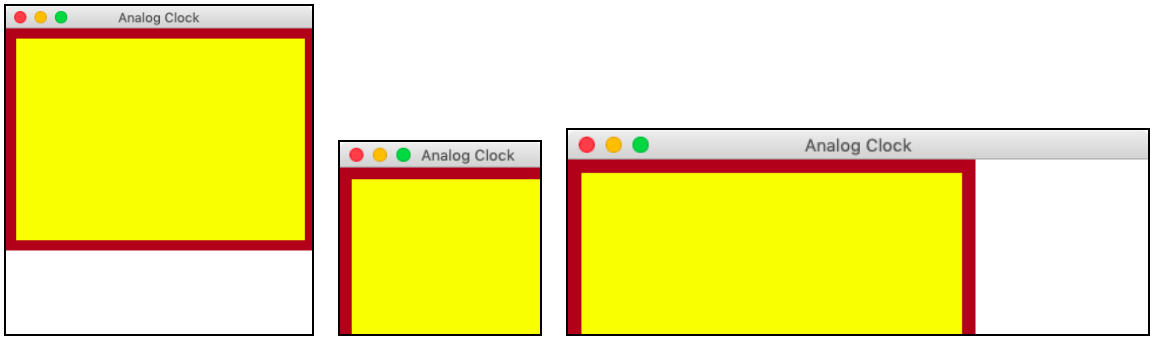
Now proceed with the following steps.

1. Preparing Frame and Canvas

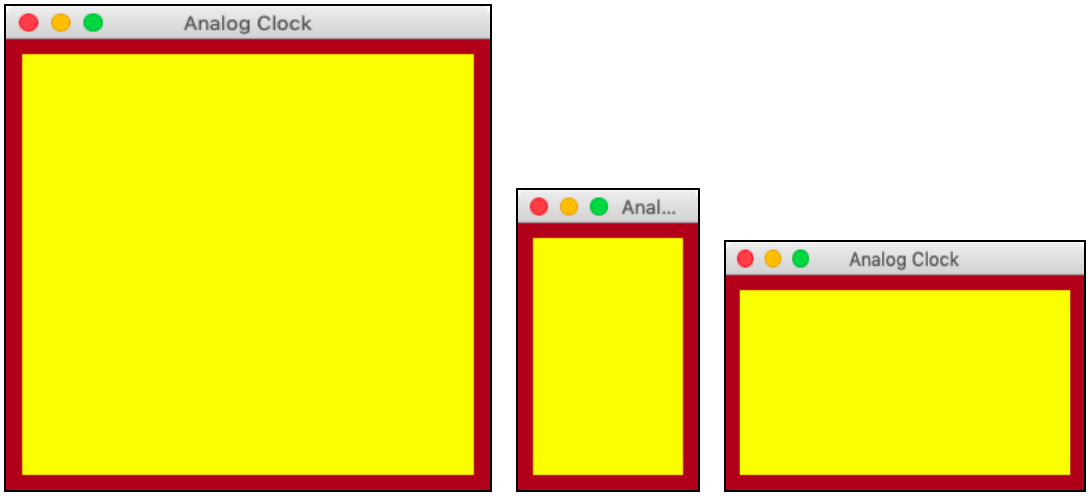
Define the `AnalogClock` class inheriting from the `Frame` class. Use the following code as your starting point. The initial code already creates a canvas inside the application's frame.

```
import tkinter as tk

class AnalogClock(tk.Frame):

    def __init__(self, parent):
        super().__init__(parent)
        self.config(bg="brown")
        self.grid(row=0, column=0)
        self.canvas = tk.Canvas(self, borderwidth=0, highlightthickness=0, bg="yellow")
        self.canvas.grid(row=0, column=0, padx=10, pady=10)


if __name__ == "__main__":
    root = tk.Tk()
    root.geometry("300x300")
    root.title("Analog Clock")
    clock = AnalogClock(root)
    root.mainloop()
```

Although the current code can create a brown frame with a yellow canvas inside, both frame and canvas do not fill the entire window and will not get resized correctly with the window.



Modify the code so that the frame and the canvas always fill the application's window, no matter how it is resized, as shown.



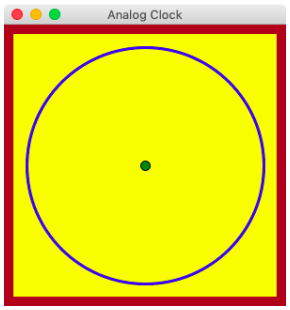
Describe the changes that you made to the original code and also explain what they do in the provided answer box.

```
import tkinter as tk


class AnalogClock(tk.Frame):

   def __init__(self, parent):
       super().__init__(parent)
       # Add rowconfigure and columnconfigure in parent and frame
       parent.rowconfigure(0,weight=1)
       parent.columnconfigure(0, weight=1)
       self.columnconfigure(0, weight=1)
       self.rowconfigure(0, weight=1)
       self.config(bg="brown")
       # Add Sticky in frame and canvas
       self.grid(row=0, column=0, sticky="NEWS")
```

```
        self.canvas = tk.Canvas(self, borderwidth=0, highlightthickness=0,
bg="yellow")
        self.canvas.rowconfigure(0, weight=1)
        self.canvas.columnconfigure(0, weight=1)
        self.canvas.grid(row=0, column=0, padx=10, pady=10, sticky="NEWS")


if __name__ == "__main__":
    root = tk.Tk()
    root.geometry("300x300")
    root.title("Analog Clock")
    # Add rowconfigure and columnconfigure in root
    root.rowconfigure(0, weight=1)
    root.columnconfigure(0, weight=1)
    clock = AnalogClock(root)
    root.mainloop()
```

2. Drawing Clock Face and Center Pin

In this step, we will draw a circular clock face with a center pin on the canvas, similar to the example below.



Let us create two ovals to represent both components in a separate `prepare_items()` method inside the `AnalogClock` class.

```
    def prepare_items(self):
        self.clock_face = self.canvas.create_oval(0, 0, 0, 0, width=3, outline="blue")
        self.center_pin = self.canvas.create_oval(0, 0, 0, 0, fill="green")
```

We then need to call this method from the class's constructor.

```
class AnalogClock(tk.Frame):
    def __init__(self, parent):
        :
        self.prepare_items()
```

For convenience, we will define two more methods. The `center_coords()` method computes and returns the current center coordinates (cx,cy) of the canvas. And the `clock_radius()` method determines the radius of the clock face, which is set to be 90% of the canvas's shorter side.

```
    def center_coords(self):
        '''
        Return a tuple (x,y) representing the current center coordinates of the canvas
        '''
        return self.canvas.winfo_width()/2, self.canvas.winfo_height()/2

    def clock_radius(self):
        '''
        Return the desired clock's radius, which is set to be 90% of the canvas's shorter side
        '''
        cx,cy = self.center_coords()
        return min(cx,cy)*0.9
```

Now we are ready to define the `update_clock()` method responsible for updating all the clock's components. At this point this method will just update the clock face and the center pin in accordance with the canvas's current size. Replace the placeholders (A) and (B) with your own code.

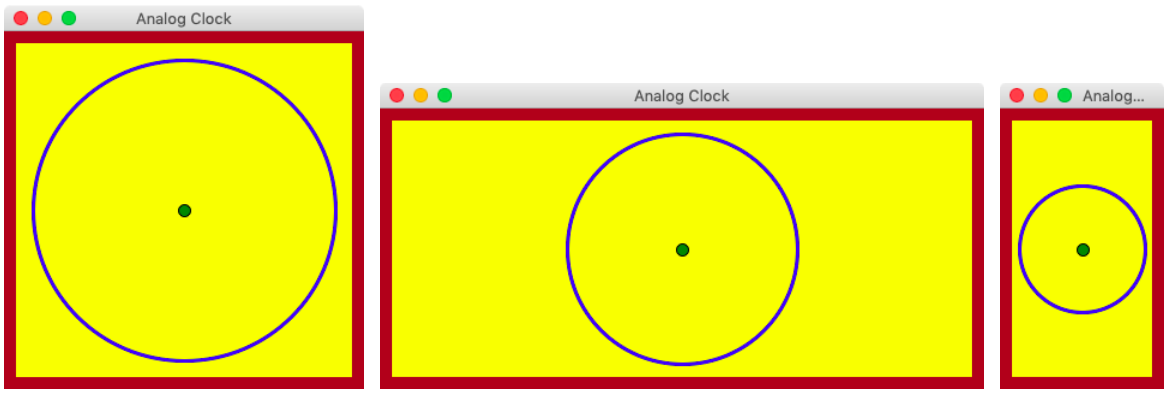
```
    def update_clock(self):
        cx,cy = self.center_coords()
        radius = self.clock_radius()

        # update the item self.clock_face so that it has the radius of self.clock_radius()
        # and is centered at coordinates (cx,cy)
        --- (A) ---

        # update the item self.center_pin so that it has the radius of 5 pixels and
        # is centered at coordinates (cx,cy)
        --- (B) ---
```

Notes: Tkinter's canvas accepts coordinates of float data type, so it's not necessary to convert them to integers.

The `update_clock()` method must be called every time the canvas's size is changed. (It must also be called every second to update the clock's hands, but that will be done later.) In the constructor, call `canvas.bind()` so that `update_clock()` is invoked whenever a ***Configure*** event is triggered

```
class AnalogClock(tk.Frame):
    def __init__(self, parent):
        :
        self.prepare_items()
        self.canvas.bind(--- your code goes here ---)
```

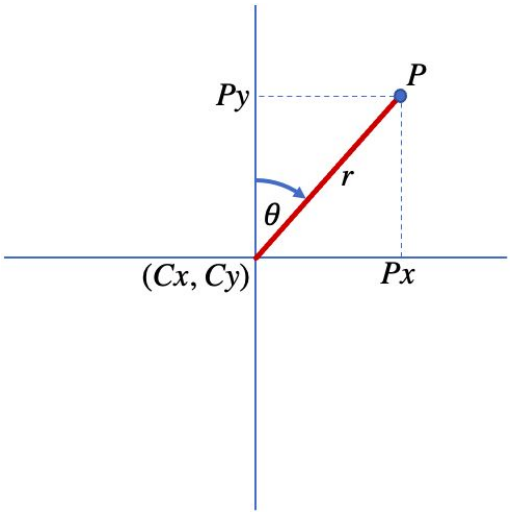
Run the code. The application's window should appear with a clock face. Try resizing the window and make sure that the size and the position of the clock face change accordingly, as shown.



3. Drawing Clock's Hands

The most important parts of our clock application are the clock's hands. Drawing them requires a little bit of mathematical and trigonometry knowledge.

From the diagram, the point P has the polar coordinates of (r, θ) centered at the point (Cx, Cy). We need to compute the corresponding Cartesian coordinates (Px, Py) for the point P in order to update the clock's hands.



The value of Px and Py can be obtained from the following equations:

$$P_x = C_x + r \cdot \sin\theta$$
$$P_y = C_y + r \cdot \cos\theta$$

The conversion formulas may look somewhat different from what you may have learned in a math class. This is because the angle θ is measured clockwise from the y-axis, not counterclockwise from the x-axis, which will suit our clock application better.

For convenience, let us define the `polar_to_canvas()` method that takes a radius and an angle theta as arguments, then returns the corresponding Cartesian coordinates (x,y) with respect to the canvas's center.

```
    def polar_to_canvas(self, radius, theta):
        '''
        Take polar coordinates (radius,theta) and return a tuple (x,y)
        representing the Cartesian coordinates on the canvas, with respect to
        the canvas's center.
        '''
        cx,cy = self.center_coords()
        x = your code goes here
        y = your code goes here
        return x,y
```

Notes on writing the `polar_to_canvas()` method

- You may want to put `import math` statement at the top of your program.
- Don't forget that `math.sin()` and `math.cos()` each takes an argument in radians, not degrees. Use `math.radians()` for the conversion.
- Recall that the y-axis in Tkinter's canvas is oriented downwards. Hence the expression to compute the value of `y` must be adjusted accordingly.

Let us extend the `prepare_items()` method to prepare three line items for the clock's hands.

```
    def prepare_items(self):
        self.clock_face  = self.canvas.create_oval(0, 0, 0, 0, width=3, outline="blue")
        self.center_pin  = self.canvas.create_oval(0, 0, 0, 0, fill="green")
        self.second_hand = self.canvas.create_line(0, 0, 0, 0, fill="gray", width=1)
        self.minute_hand = self.canvas.create_line(0, 0, 0, 0, fill="black", width=4)
        self.hour_hand   = self.canvas.create_line(0, 0, 0, 0, fill="black", width=8)
```

We also need to extend the `update_clock()` method to update all these lines. The code below only demonstrates the update of the "second" hand, whose length is 90% the clock's radius.

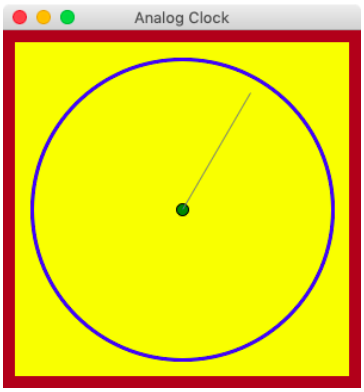
```
    def update_clock(self):
        cx,cy = self.center_coords()
        radius = self.clock_radius()
        :
        now = datetime.now()
        sec_angle = now.second/60*360
        x,y = self.polar_to_canvas(radius*0.9, sec_angle)
        self.canvas.coords(self.second_hand, cx, cy, x, y)
```

The code above also assumes that the statement

```
from datetime import datetime
```

has been added to the top of the program.

The clock should now have its "second" hand show up. However, as the hand is updated only when the `update_clock()` method is called, the hand will not be moving unless the window is resized.



Complete the `update_clock()` method so that it properly displays the "minute" and "hour" hands. Remember that the minute and hour hands must also move continuously, though more slowly than the second hand. Explain what you have done in the box below.

```
def update_clock(self):
   cx, cy = self.center_coords()
   radius = self.clock_radius()

   # update the item self.clock_face so that it has the radius of
self.clock_radius()
   # and is centered at coordinates (cx,cy)
   self.canvas.delete(self.clock_face)
   self.clock_face = self.canvas.create_oval(cx + radius, cy - radius, cx -
radius, cy + radius, width=3,
                                             outline="blue")

   # update the item self.center_pin so that it has the radius of 5 pixels and
   # is centered at coordinates (cx,cy)
   self.canvas.delete(self.center_pin)
   self.center_pin = self.canvas.create_oval(cx + 5, cy - 5, cx - 5, cy + 5,
fill="green")

   now = datetime.now()
   # Second
   sec_angle = now.second / 60 * 360
   x, y = self.polar_to_canvas(radius * 0.9, sec_angle)
   self.canvas.coords(self.second_hand, cx, cy, x, y)
   # Minute
   min_angle = now.minute / 60 * 360
   x, y = self.polar_to_canvas(radius * 0.9, min_angle)
   self.canvas.coords(self.minute_hand, cx, cy, x, y)
   # Hour
   if now.hour > 12:
       hour = now.hour-12
   else:
       hour = now.hour
   hour_angle = hour / 12 * 360
   x, y = self.polar_to_canvas(radius * 0.7, hour_angle)
   self.canvas.coords(self.hour_hand, cx, cy, x, y)
```

4. Making the Clock Tick

The final step is to make the program call the `update_clock()` method every second. Explain what you have done in the box below.

```
def update_clock(self):
   cx, cy = self.center_coords()
   radius = self.clock_radius()

   # update the item self.clock_face so that it has the radius of
self.clock_radius()
   # and is centered at coordinates (cx,cy)
   self.canvas.delete(self.clock_face)
   self.clock_face = self.canvas.create_oval(cx + radius, cy - radius, cx -
radius, cy + radius, width=3,
                                             outline="blue")

   # update the item self.center_pin so that it has the radius of 5 pixels and
   # is centered at coordinates (cx,cy)
   self.canvas.delete(self.center_pin)
   self.center_pin = self.canvas.create_oval(cx + 5, cy - 5, cx - 5, cy + 5,
fill="green")

   now = datetime.now()
   # Second
   sec_angle = now.second / 60 * 360
   x, y = self.polar_to_canvas(radius * 0.9, sec_angle)
   self.canvas.coords(self.second_hand, cx, cy, x, y)
   # Minute
   min_angle = now.minute / 60 * 360
   x, y = self.polar_to_canvas(radius * 0.9, min_angle)
   self.canvas.coords(self.minute_hand, cx, cy, x, y)
   # Hour
   if now.hour > 12:
       hour = now.hour-12
   else:
       hour = now.hour
   hour_angle = hour / 12 * 360
   x, y = self.polar_to_canvas(radius * 0.7, hour_angle)
   self.canvas.coords(self.hour_hand, cx, cy, x, y)
   # Add this line to make function update every seconds
   self.after(1000, self.update_clock)
```

Warning: do not attempt to schedule a timer inside the `update_clock()` method. Doing so would cause a new timer to be scheduled every single time the application window gets resized.

Submitting Your Work

- Name your application code `analog-clock.py` and submit it along with this labsheet.
- Don't forget to click **Turn In** to complete your submission.