The goal of the project is to apply client/server computing to draw the Mandelbrot set, via a very simple GUI:

The Mandelbrot set is defined by an iterative algorithm. A point in the plane with coordinates \((x, y)\) is represented by the complex number \(c = x + iy, i = \sqrt{-1}\). Starting at \(z = 0\), count the number of iterations of the map \(z \mapsto z^2 + c\) takes to produce values with modulus larger than 2. This number of iterations determines the inverted grayscale of the pixel with coordinates \((x, y)\) in the plot of the Mandelbrot set, for \(x \in [-2, 0.5]\) and \(y \in [-1, +1]\). The canvas on the left (400 pixels high and 500 pixels wide) needs 16,652,580 iterations.

On larger or smaller screens we may take more or fewer pixels, but we must observe the ratio 5/4 between width and height, otherwise the Mandelbrot set will be distorted.

Grayscales are encoded by one byte, as a hexadecimal number between 0 and 255 (0 is black and 255 is white). To invert a grayscale \(k\), compute \(255 - k\). Points close to \((0, 0)\) never grow large under the map \(z \mapsto z^2 + c\) and since we have only 256 grayscales, the iterations should stop when the iteration count reaches 255.

In Tkinter there is no command to draw one pixel on screen. Instead we draw a square with length of the side equal to one. For a canvas widget \(cv\), to color the \((i, j)\)-th pixel, this command can take the form

\[
\text{cv.create_rectangle}(i,j,i+1,j+1,width=1,outline=\text{code},fill=\text{code})
\]

where the \text{code} is a string encoding the grayscale in RGB format. On the course web site is a simple GUI (in the script \text{grayscale.py}) to display all 256 shades of gray.

Although we are tempted to build a full fledged GUI to explore the wonderful fractal nature, we will not do it — although it is not prohibited to add more features.

Because the drawing of the Mandelbrot set is computationally intensive we distribute the iterations among clients. When the server script is launched, the user is first prompted to enter the number of threads in the multithreaded server. After the user has entered the number of threads, the simple GUI is launched. As shown above, this simple GUI only has one canvas and a \text{draw} button. When the user of the GUI presses this \text{draw} button, the server threads are started.
For example, starting the server script at the command prompt:

```
$ python mandelserver.py
give #threads : 5
server is ready for 5 clients
server starts 5 threads
```

The message `server starts 5 clients` is displayed after the user pressed the `draw` button of the GUI. If after that, there is only then one client running, we see:

On the left we see one fifth of the Mandelbrot set drawn with data computed by one client. If the server has $n$ handler threads, the $k$th thread will send to its client slice coordinates $k + n \times m$, for $m = 0, 1, \ldots$, and collect from its client a list of inverted grayscales defining one slice of the drawing, labeled with coordinates $k + n \times m$. For example, with five threads, the third client will compute the data for slices 2, 7, 12, and so on. If the canvas is 400 pixels high, each time the client sends a list of 400 numbers to its handler thread. When the client receives -1 it stops.

The user can launch client scripts at any time, up to the total amount of handler threads provided by the server. To monitor the progress of client and server, print out messages with identifying information about which slice is computed, for example, the dialogue for the slices in the plot above:

```
$ python mandelclient.py
client is connected
client received 0
computing slice 0
drawing slice 0
client received 5
computing slice 5
```

At the left we see the messages displayed by the client, while the right continues the messages displayed by the server. As the user launches more client scripts, the canvas will show more slices of the Mandelbrot set.

Note that by distributing the computations, the total amount of time it takes to display the Mandelbrot set is actually not reduced because of the added overhead of thread scheduling and network communications, but we will see an outline of the Mandelbrot set faster in the form of its slices.
Some important points:

1. Since this project consists in two parts, you may collaborate in pairs. Both authors will receive the same number of points.

2. This project combines GUI development, multithreading, and client/server computing. Before integrating these three different technologies, make sure the different pieces work. For example, the simple GUI and client/server computation of the grayscale pictures can be developed separately.

3. Use a clear functional design, avoiding extremely long sequences of code in one function. Use classes to implement the GUI and the multithreaded server. Provide appropriate documentation, i.e.: every function must have a documentation string.

4. Handing in an incomplete but working program is better than handing in a program that crashes or does not run at all.

5. The solution consists of two files: one file with the code for the server and another with code for the client.

6. The first line of your Python programs must be

   # MCS 275 Project Four by <Authors>

   where you replace the <Authors> by your names.

7. Email your solution to the project to jan@math.uic.edu before 11AM on Wednesday 16 April so the date of the email is proof of an on time submission. As a backup, bring also a printed version of your solution to class.

If you have questions or difficulties with the project, feel free to come to my office for help.