CSCI 1133, Fall 2017 Due: 7:00 am, Monday, Dec 11
Programming Examination 3

PLEASE NOTE: This is a graded assessment of individual programming understanding and ability, and is ***not*** a collaborative assignment; you must design, implement and test the solution(s) completely on your own without outside assistance from anyone. You may not consult or discuss the solution with anyone. In addition, you may not include solutions or portions of solutions obtained from any source other than those provided in class. Note that *providing* a solution or assisting another student in any way on this examination is also considered academic misconduct. Failure to heed these directives will result in a failing grade for the course and an incident report filed with the Office for Student Conduct and Academic Integrity for further sanction.

A. (100 points) **Mandelbrot Fractals**

This examination consists of designing and writing a single well-structured Python program that must be committed and pushed to your remote GitHub examination repository *prior* to the deadline.

Late submissions will not be accepted and will result in a zero score for the exam.

TA help for this examination will not be provided. If you have clarification questions, they must be addressed to the graduate TA for the class.

The total point value will be awarded for solutions that are *complete*, *correct*, and *well structured*. A *"well structured"* program entails good design that employs meaningful functional decomposition, appropriate comments and general readability (descriptive names for variables and procedures, appropriate use of blank space, etc.) If you are not sure what this means, review the "well-structured" program requirements provided in Lab2.

Note that your work will be graded using, and must function correctly with, the current version of Python 3 on CSE Labs UNIX machines. If you complete this programming exam using a different system, it is *your* responsibility to ensure it works on CSELabs machines prior to submitting it.

The rubric includes the following specific (accumulative) point deductions:

• Missing academic integrity pledge	-100 points
• Syntax errors	-50 to -100 points
• Misnamed source file or incorrect repository	-25 points
• Use of global variables	-25 points
• Missing main program function	-25 points

Examination Repository

Examination files must be submitted to GitHub using your remote exam repository. Exam repositories have already been created for each registered student and are named using the string `exam-` followed by your X500 userID (e.g., `exam-smit1234`). You must first clone your exam repository in your local home directory before submitting the materials for this exam. If you are having difficulty, consult the second Lab from earlier in the semester or the GitHub tutorial on the class webpage. If your exam repository is missing or something is amiss, please contact the graduate TA. DO NOT SUBMIT YOUR EXAM FILES TO YOUR LAB/EXERCISE REPOSITORY!

Introduction

For this project, you will employ basic Object Oriented Programming techniques to construct a program that displays a *Mandlebrot* fractal image using Turtle graphics. The program will allow the user to "click" anywhere on the fractal image to "zoom-in", demonstrating the self-similarity of complex fractals. The program will consist of three separate classes that interact to construct and display a *Mandelbrot* fractal.

Background

Fractals are mathematical sequences that are "made of parts similar to the whole in some way". When we graph the values of complex fractal sequences on an *x,y* Cartesian plane, they often produce stunning visual images. Check out the following Wikipedia description:

https://en.wikipedia.org/wiki/Fractal

Enlarging ("zooming-in") on fractal images will often reveal "self-similar" patterns, i.e., small patterns that look exactly like the original.

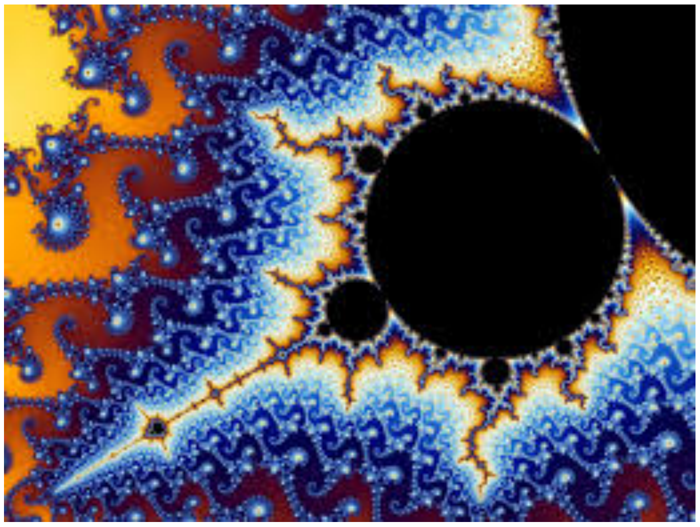
One famous fractal image is derived from the so-called *Mandelbrot set* which is based on a curious property exhibited by *Mandelbrot* sequences. A *Mandelbrot sequence* is constructed by starting with an arbitrary *complex* number, C_0. Each subsequent value of the sequence is computed from its predecessors according to the following equation:

$$Z_0 = C_0$$
$$Z_{n+1} = (Z_n)^2 + Z_0$$

For an arbitrary complex number C_0, the *Mandelbrot* sequence is an infinite series in which the first value is C_0, the next value is C_0 added to the square of the previous value in the sequence, and so on.

The values in most *Mandelbrot* sequences will grow infinitely. However there is a special subset of the starting values, C_0 that result in *Mandelbrot* sequences that remain bounded within a finite region! The collection of all such starting values, C_0, is referred to as the *Mandelbrot set*. The sequences formed from the starting values in the *Mandelbrot set* are all bounded by a finite region in the complex plane and only contain complex numbers whose absolute values less than 2.

When plotted on an *x,y* Cartesian plane, the *Mandelbrot* set produces stunning fractal patterns such as this one (from wikipedia.org):



Drawing Mandelbrot Fractals

We can visualize the *Mandelbrot* set by graphing it on the complex plane instead of the real plane. Throughout the semester we've been using Turtle graphics to graph numbers in the real plane. If we passed two real numbers to `turtle.goto(x, y)` the turtle would go the position x on the x-axis and the position y on the y-axis. The complex plane is similar to the real plane, however we no longer think in terms of *x* and *y* axes. Instead, we will let the horizontal axis correspond to the *real* part of a complex number and the vertical axis correspond to the *imaginary* part of a complex number.

In this way, we can match each of the *x,y* turtle coordinates on a screen to an equivalent point on the complex plane and then test each complex point for membership in the *Mandelbrot* sequence. If the complex point represents a starting value, C_0 for a Mandelbrot sequence that always produces a number whose absolute value is <= 2, then the point is a member of the *Mandelbrot* set and colored black. If you attempt to compute the *Mandelbrot* sequence and you encounter a complex number with an absolute value greater than two, then you color that point white.

Note that we cannot compute an entire (infinite) *Mandelbrot* sequence, because we cannot run our computers for an infinite amount of time! Instead, when drawing fractals we compute the *Mandelbrot* sequence up to a finite limit (50 iterations will be a good approximation for this project). If the *Mandelbrot* sequence for a given value of C_0 goes out of bounds (absolute value >2) before this limit is reached, then C_0 is not a member of the set. If all of the values up to Z_{limit} have absolute values <= 2, then we will assume that C_0 *is* a member of the *Mandelbrot set*.

We now have all of the pieces to plot a *Mandelbrot set* in black and white. We visit each *x,y* coordinate on the turtle canvas, translate that coordinate into a coordinate on the complex plane, then use that complex number as the starting value to compute a *Mandelbrot* sequence until we reach the iteration limit. At that point, if the complex coordinate is still in the *Mandelbrot set* we could color it black. If we found that the coordinate was not in the Mandelbrot set, we could color that coordinate white.

Adding color

Black and white fractals are nice, but it's even more interesting to plot a color representing the *proximity* of a complex point to the *Mandelbrot set*. Since the *Mandelbrot sequence* is a sum of squares, a point that quickly generates an out-of-bounds complex number should be colored differently than a point that generates an out-of-bounds value on the last iteration.

Because color is not an intrinsic part of the *Mandelbrot set*, there are many ways we could color our image. One way is to create a list of some fun colors and indicate how close a point is to membership in the *Mandelbrot set* by choosing a color based on the index of the first term in the sequence whose absolute value is > 2.

Part 1: Complex Number Class

In order to create a Mandelbrot sequence, we must be able to represent complex values and compute complex expressions. Although Python provides a built-in numeric type: `complex` that supports operations on complex numbers, assume that such a class does not exist and create your own user-defined complex number class to use in Part 2. Note that a *complex number* is of the form $a + bi$ where a and b are real numbers and $i^2 = -1$. For example, $2.4 + 5.2i$ and $5.73 - 6.9i$ are *complex* numbers. Here, a is called the *real* part of the complex number and bi the *imaginary* part. For more information on complex numbers refer to the Wikipedia page:

https://en.wikipedia.org/wiki/Complex_number

Create and (thoroughly test!) a class named `Complex` to represent complex numbers. Save your class definition in a module named `complex.py`. The members of this class are as follows:

Instance variables:

- Private `real` and `imag` of type `float` to represent the complex number `real` + `imag` *i*

Methods:

- A constructor that will initialize `real` and `imag` from user supplied arguments (default to zero)
- An overloaded __repr__ method that will return the complex value as a string in this form, e.g.:

  ```
  2.3 + 4.6i
  ```

  Note that if the imaginary component is zero, omit the "+ xi" part (i.e., *display* the number as a non-complex "float", and if the imaginary component is negative, replace the '+' with '-'.
- Accessor and mutator methods for both instance variables
- An overloaded __add__ (addition) operator that will return the sum of two `Complex` objects. For complex numbers a + b*i* and c + d*i*, addition is defined as: (a+c) + (b+d)*i*.
- An overloaded __mul__ (multiplication) operator that will return the product of two `Complex` objects. Note that the product of 2 complex numbers is determined using distributed multiplication (as in multiplying two polynomials):

$$\begin{aligned}(2 + 3i) \bullet (4 + 5i) &= 2(4 + 5i) + 3i(4 + 5i)\\ &= 8 + 10i + 12i + 15i^2\\ &= 8 + 22i + 15(\mathbf{-1})\\ &= -7 + 22i\end{aligned}$$

- An overloaded __abs__ (absolute value) operation that will return the absolute value of a complex number. Note that the absolute value is the *distance* from the origin of the complex plane (0 + 0*i*) to the complex number (HINT: "pythagoras").

Part 2: Mandelbrot Sequence Class

After you have your complex number class working (and tested!), construct a separate class to represent *Mandelbrot* sequences. The class name is `Mandelbrot` and should contain the following members:

Instance variables:

- Private `limit` (type `int`) that represents the maximum length *Mandelbrot* sequence
- Private `colormap`, a list of strings containing the Turtle color values used to color each point
- Private `cardinality` (type `int`) that indicates the number of *consecutive* complex values in the computed sequence (starting from 0) whose absolute value is "in bounds" (i.e., <= 2)

Methods:

- A constructor that will take two arguments: the starting value, C_0 for the *Mandelbrot* sequence (type `Complex`) and the maximum length sequence to be computed (type `int`, defaults to 50). The constructor should compute the Mandelbrot sequence for the given starting value and initialize the instance variables.
- An accessor method named `get_color` that will return a color from the color map corresponding to the cardinality of the set. Lower cardinality values (further from the *Mandelbrot* set) will correspond with colors earlier in the list. A cardinality value equal to the maximum (`limit`) should correspond to the color '`black`'.

Part 3: Graphical Display Class

After you have your `Mandelbrot` class working (and tested!), construct a third separate class named `Display` to handle the graphical display details. The `Display` class should instantiate a turtle object and handle on-click operations. Your class must include the following:

- A constructor method that takes no arguments and handles hiding the turtle, setting the turtle speed to zero, changing the \`tracer\` so that the program can draw more quickly, setting up the on-click action, and drawing the first *Mandelbrot* fractal at the default zoom. The default zoom should draw the complex point -2 - 2*i* in the bottom left corner, and the complex point 2 + 2*i* in the top right corner. You need to include instance variables to keep track of the *bounds* of the visible part of the *Mandelbrot set* and subsequently update them as the fractal is zoomed.
- A mutator method named `click`: that is called whenever the user clicks on the screen. (On-click actions are described in more detail in the Python 3 Turtle documentation). Note: if a *Mandelbrot set* is currently being drawn, or if the x,y coordinates are outside the fractal image, this method should do nothing. If the turtle is not currently drawing the Mandelbrot set, the zoom operation should be performed and the fractal redrawn.
- A mutator method named `zoom` that modifies the complex plane coordinates. This method should modify the complex plane coordinates and redraw the *Mandelbrot set* so that it is magnified 2x and re-centered around the point the user clicked. This will halve the width and height of the viewable section of the *Mandelbrot set*. In order to accomplish this you should determine which point in the complex plane was clicked, then recalculate the boundaries of the viewable section of the plane by making that point the *center* of the new grid and making the new width and height of the grid half of the old width and height.
- A mutator method named `draw` that will re-draw the *Mandelbrot* fractal at the current resolution. This method should visit each *x, y* coordinate on the turtle canvas, translate that coordinate onto the (zoomed) section of the complex plane, and color it according to how close it is to membership in the *Mandelbrot set* (see suggestion 1 below). Note that computing membership in the *Mandelbrot set* for each pixel on the canvas can *significantly* slow down your computer. Limiting the length of the sequence to 50 iterations will speed it up, but in order to draw your fractal as quickly as possible, it is recommended that you do **NOT** accomplish the drawing using the `.dot` or `.stamp` methods. Instead, color each pixel (*x, y*) by putting the pen down, changing the pen to use the appropriate color, and moving the turtle to a pixel close by (e.g., *x, y* + 1), which will paint the canvas with approximately one pixel worth of color.

General Program Requirements

Using Turtle graphics, write a well-structured Python program that will display the fractal *Mandelbrot set*. Your program will consist of 3 class definitions as described and a single `main()` function. The main function simply needs to instantiate a single `Display` object to display the interactive graphics.

Your program must do the following:

- Include the academic integrity pledge in the program source code (details below)
- Use turtle graphics to implement the graphical interface (set the turtle speed to 0 in order to speed things up, and set the pensize to 1 for maximum resolution).
- Your program should have a main function as described in class. Running your program from the command line should automatically draw the Mandelbrot set at the default zoom and set up the screen to accept user clicks. Note that the main function simply needs to instantiate a `Display` object.

Submission

Your `Complex` and `Mandelbrot` classes should each reside in separate modules named `complex.py` and `mandelbrot.py` respectively. Your main program should include a `main()` function and reside in a module named `fractal.py` that includes your `Display` class. Commit and push all three modules to your exam repository. Please be sure to observe the naming requirements.

Constraints:

- You may use any built-in Python object class methods (string, list, etc.) except the built-in complex number class
- You may use *imported* functions and class methods from the `turtle` module only.
- Do not use the `.setworldcoordinates` turtle method. Use the default (cartesian) settings for the turtle screen (0,0 at the center of the window)

Academic Integrity Pledge

The first lines of your program *must* contain the following text (exactly as it appears below), replacing the first line with your full name and X500 ID. If this statement does not appear in your program source, you will receive a score of zero for the exam:

```
# <replace this line with your name and x500 ID (e.g., John Smith smit1234>

# I understand that this is a graded, individual examination that may not be
# discussed with anyone.  I also understand that obtaining solutions or
# partial solutions from outside sources, or discussing any aspect of the exam
# with anyone is academic misconduct and will result in failing the course.
# I further certify that this program represents my own work and that none of
# it was obtained from any source other than material presented as part of the
# course.
```

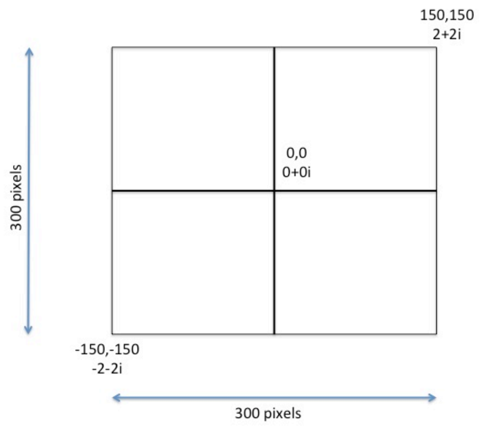
Suggestions:

1) We suggest that you provide the image size (in turtle "units") to the Display class constructor as an argument. In this way, you can debug things at a smaller size and then use a larger (higher resolution) to display the fractal when thing are working.

2) Make sure to set the turtle speed to 0, pensize to 1, and hide the turtle. You will also need to use the `.tracer` method to speed things up. `.tracer(2000,0)` seems to work reasonably well, but you can experiment with the value.

3) The process of plotting employs a nested loop that systematically moves (use the `.goto` method) the turtle to every *x,y* turtle coordinate of the display. As the turtle moves with the pen down, it will draw a small (1 or 2 pixel) line in the color specified by the colormap for the *Mandelbrot* sequence. Note that the turtle must be moved in *x,y turtle* coordinates, but the *Mandelbrot* sequence for each *x,y* location must be computed in *complex* number plane coordinates. This requires a conversion from turtle coordinates (*x,y*) to complex plane coordinates (*real, imaginary*).

Consider the example figure below for a 300 by 300 pixel turtle window. The center coordinates of the window are 0,0 which correspond to 0+0i in the complex plane (for the initial display). The lower leftmost pixel is at -150, -150 which corresponds to the -2-2i in the complex plane.

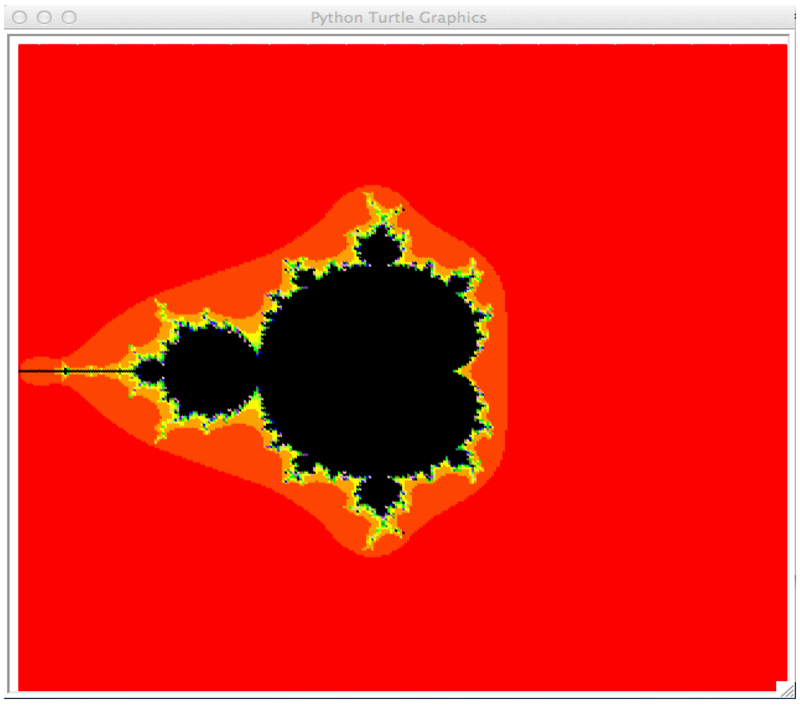


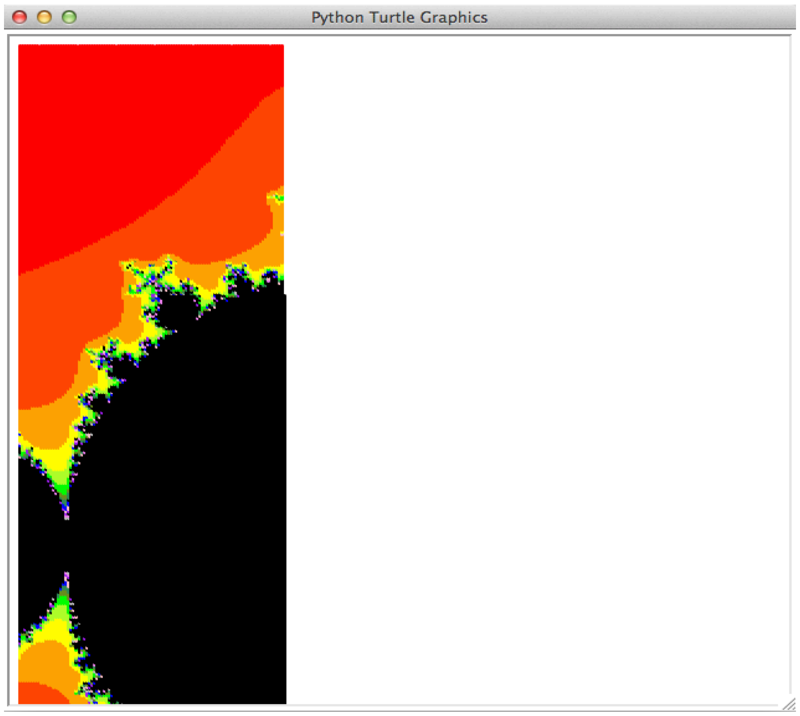
The conversion is straightforward based on the following scale relationship (a similar relationship holds for the vertical axis):

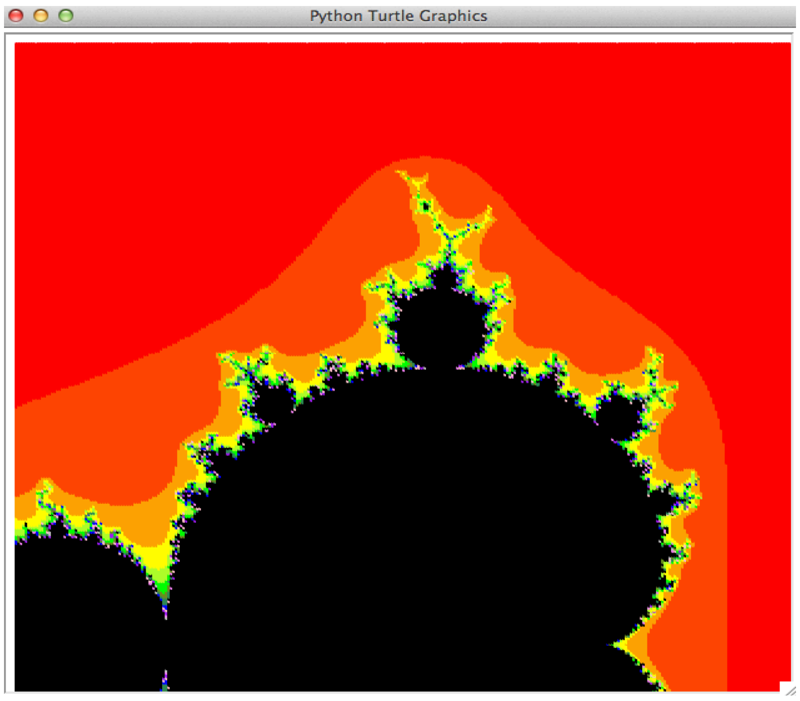
$$x / (x_{max} - x_{min}) = real / (real_{max}-real_{min})$$

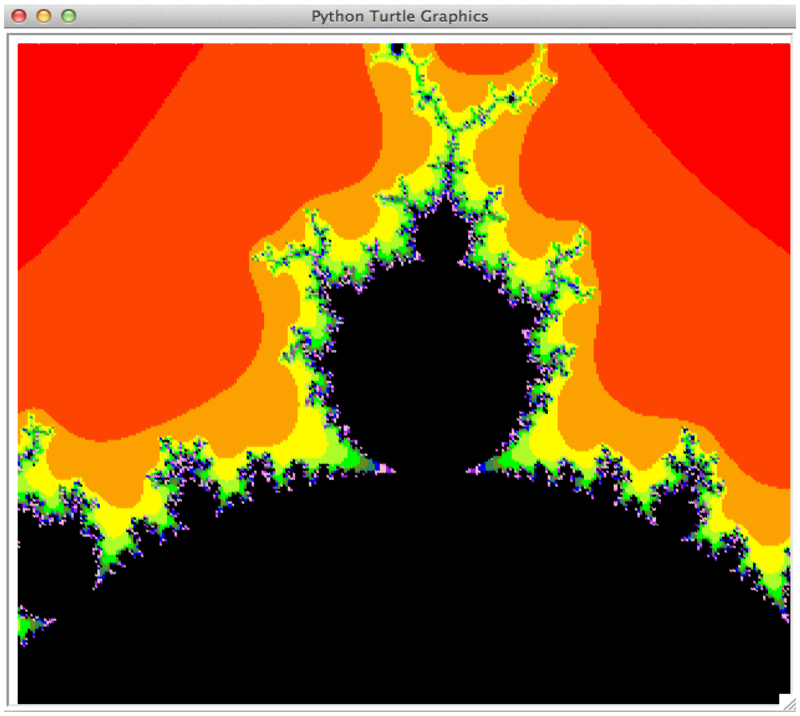
Note that after a "zoom" operation, the *x,y* plane will remain unchanged, however the complex plane will be rescaled and the origin will be offset from 0,0 relative to the image!

Example Program Output:




Python Turtle Graphics


Python Turtle Graphics


Python Turtle Graphics