HW5: Picturing the Mandelbrot Set

The Mandelbrot Set

In this homework you will build a program to visualize and explore the points in and near the Mandelbrot Set. In doing so, you will have the chance to:

- Use loops and nested loops to solve *complex* problems
- Develop a program using incremental design (starting with a simple task and gradually adding levels of complexity)
- Connect with mathematics and other disciplines that use fractal modeling

Introduction to `for` Loops!

To build some intuition about loops, first write two short functions:

- Multiplication is just repeated addition! Write a function named `mult(c,n)` that returns the product c times n, but without multiplication. Instead, start a `result` at 0 and repeatedly *add* `c` into that `result`. It must use a `for`.
  Here are a couple of cases to try:

  ```
  mult(6, 7) == 42

  mult(1.5, 28) == 42.0
  ```

- The next function will build the basic Mandelbrot update step, which is `z = z**2 + c` for some constant `c`.

  To that end, write a function named `update(c, n)` that starts a new value, `z`, at zero, and then repeatedly updates the value of `z` using `z = z**2 + c` a total of `n` times. In the end, the function should return the final value of `z`. The value of `n` will be a positive integer. Here are the `def` line and docstring to get you started: Here are a couple of cases to try:

  ```
  update(1, 3) == 5


  update(-1, 3) == -1

  update(1, 10) ==
  37918623102659260828682350280278932773702331522473885847617341507177682
  54410341175325352026
  ```

```
update(-1, 10) == 0
```

You'll use these ideas (by a variant of `update`) in building the Mandelbrot set.

Introduction to the Mandelbrot Set

The *Mandelbrot set* is a set of points in the complex plane that share an interesting property that is best explained through the following process:

- Choose a complex number *c*.
- With this *c* in mind, start with $z_0 = 0$.
- Then repeatedly iterate as follows:
- $z_{n+1} = z_{n2} + c$

The Mandelbrot set is the collection of all complex numbers *c* such that this process does ***not*** diverge to infinity as `n` gets large. In other words, for some *c*, if z_n diverges to infinity (becomes unreasonably large) then *c* is *not* in the set; otherwise, it is.

The Mandelbrot set is a *fractal*, meaning that its boundary is so complex that it cannot be well approximated by one-dimensional line segments, regardless of how closely one zooms in on it. Fractals are cool!

The `inMSet` function

The next task is to write a function named `inMSet(c, n)` that accepts a complex number `c` and an integer `n`.

This function will return a Boolean:

- `True` if the complex number `c` is probably in the Mandelbrot set and
- `False` otherwise.

First, we will introduce Python's built-in support for complex numbers.

Python and complex numbers

In Python a complex number is represented in terms of its real part `x` and its imaginary part `y`. The mathematical notation would be *x* + *y*i, but in Python the imaginary unit is typed as `1.0j` or `1j`, so that:

```
c = x + y*1j
```

would set the variable `c` to the complex number with real part `x` and imaginary part `y`.

Doing `x + yj` does not work, because Python thinks you're using a variable *named* `yj`.

Also, the value `1 + j` is not a complex number: Python assumes you mean a variable named `j` unless there is an int or a float directly in front of it. Use `1 + 1j` instead.

Side note: Mathematicians use *i*; engineers use *j* because in engineering, *i* refers to electrical current. Python follows the engineering convention.

Try it out: Just to get familiar with complex numbers, at the Python prompt try

```
In [1]: c = 3 + 4j

In [2]: c
Out[2]: (3+4j)

In [3]: abs(c)
Out[3]: 5.0

In [4]: c**2
Out[4]: (-7+24j)
```

Python is happy to use the power operator (**) and other operators with complex numbers. However, note that you cannot compare complex numbers directly—they are 2d points, so there's no "greater than"! Thus, you cannot write c > 2 for a complex c (It will crash with a TypeError).

You CAN compare the magnitude, however: `abs(c) > 2`. Note that the built-in `abs` function returns the magnitude of a complex number.

Thinking about the `inMSet` function:

To determine whether a number *c* is in the Mandelbrot set, you will

- Start with $z_0 = 0 + 0j$ and then
- Repeatedly iterate $z_{n+1} = z_{n2} + c$

to see if this sequence of z_0, z_1, z_2, etc. stays bounded.

To put it another way, we need to know whether the magnitude of these z_k go off toward infinity.

Truly determining whether this sequence goes off to infinity isn't feasible. To make a reasonable guess, we will have to decide on two things:

- The number of times we are willing to wait for the $z_{n+1} = z_n^2 + c$ process to run
- A value that will represent "infinity"

We will run the update process n times. The n is the second argument to the function inMSet(c, n). This is a value you will experiment with, but 25 is a reasonably good value.

Infinity can be surprisingly small![Citation Needed] It has been shown that if the absolute value of a complex number *z* ever gets larger than 2 during the update process, then the sequence will *definitely* diverge to infinity.

There is no equivalent rule that tells us that the sequence definitely *does not* diverge, but it is *very likely* it will stay bounded if abs(z) does not exceed 2 after a reasonable number of iterations, and n is that "reasonable" number, starting at 25.

Writing inMSet

You should **copy** your update function and change its name to inMSet.

It's definitely better to copy and change that old function—*do **not** call update directly.* Please don't.

To get you started, here is the first line and a docstring for inMSet:

```
def inMSet(c, n):
    """inMSet accepts
            c for the update step of z = z**2+c
            n, the maximum number of times to run that stepThen, it
        returns
            False as soon as abs(z) gets larger than 2 True if
            abs(z) never gets larger than 2 (for n
iterations)"""
```

The inMSet function should return False if the sequence $z_{n+1} = z_n^2 + c$ ever yields a *z* value whose magnitude is greater than 2. It returns True otherwise.

Note that you will ***not*** need different variables for *z_0*, *z_1*, *z_2*, and so on. Rather, you'll use a single variable `z`. You'll update the value of `z` within a loop, just as in `update`.

Also, note that the loop should return `False` ***as soon as*** the value of `z` exceeds 2. If you try to wait until the end of the loop, you may wind up with a complex number that is too big for Python to represent!

Make sure that you are using `return False` somewhere *inside* your loop. You will want to `return True` ***after*** the loop has finished all its iterations!

Check your `inMSet` function by copying and pasting these examples:

```
In [2]: inMSet(0 + 0j, 25) == True # this one is in the set

# this one below is NOT in the set
# WARNING: this one might freeze or crash Python UNLESS your function
# returns False *as soon as* the magnitude is larger than 2.
# In other words, you need to return False _inside_ the loop
# (inside your if or your else, whichever is appropriate)!

inMSet(3 + 4j, 25) == False

inMSet(0.3 + -0.5j, 25) == True

inMSet(-0.7 + 0.3j, 25) == False

inMSet(0.42 + 0.2j, 25) == True

inMSet(0.42 + 0.2j, 50) == False
```

Getting too many `True`s?

If so, you might be checking for `abs(z) > 2` *after* the `for` loop finishes. Besure to check *inside* the loop!

There is a subtle reason you need to check inside the loop:

Many values get so large so fast that they overflow the capacity of Python's floating-point numbers. When they do, *they cease to obey greater-than / less-than relationships*, and so the test will fail. The solution is to check whether the magnitude of `z` ever gets bigger than 2 ***inside*** the for loop, in which case you should immediately `return False`.

The `return True`, however, needs to stay outside the loop!

As the last example illustrates, when numbers are close to the boundary of the Mandelbrot set, many additional iterations may be needed to determine whether they escape. This is why it is so computationally intensive to build high-resolution images of the Mandelbrot set.

Creating images with Python

Getting started

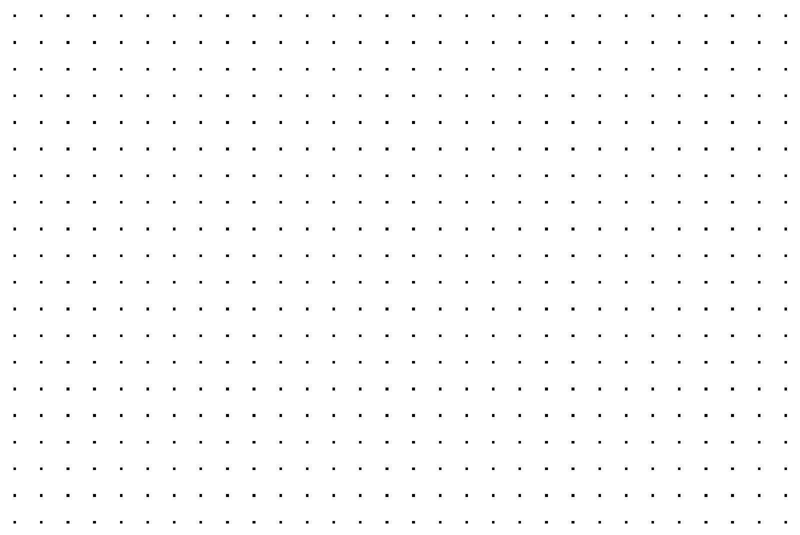
Try out this code if you need additional help with how images work. This will not be graded at all! This is just for practice with images.

```
def weWantThisPixel(col, row):
    """This function returns True if we want to showthe pixel at
       col, row and False otherwise."""
    if col % 10 == 0   and   row % 10 == 0:
        return True
    else:
        return False

def test():
    """This function demonstrates howto
       create and save a PNG image.
    """
    width = 300
    height = 200
    image = PNGImage(width, height)

    # create a loop in order to draw some pixelsfor col in
    range(width):
        for row in range(height):
            if weWantThisPixel(col, row):
                image.plotPoint(col, row)
    # we looped through every image pixel; we now write the file
    image.saveFile()
```

Save this code, and then run it by typing test(), with parentheses, at the Python shell.

If everything goes well, test() will run through the nested loops and print a message that the file test.png has been created. That file should appear in the same directory as where you ran the file.

For the above function, it should be all white except for a regular, sparse point field, plotted wherever the row number and column number were both multiples of 10:



An image thought-experiment to consider...

Consider what might happen in the code above, this change took place:

if col % 10 == 0 and row % 10 == 0:
to the line
if col % 10 == 0 or row % 10 == 0:

Then, make that change from and to or and try it. You may need to open the image again.

Some notes on how the test function works...

There are three lines of the test function that warrant a closer look:

- image = PNGImage(width, height) This line of code creates a variable of type PNGImage with the specified height and width. The image variable holds *the whole image*! This is similar to the way a single variable—often called L—can hold an arbitrarily large list of items. When information is gathered together into a list or an image or another structure, it is called a *software object* or just an *object*.

We will build objects of our own design in a couple of weeks; this Homework is an opportunity to use them without worrying about how to create them from scratch.

- image.plotPoint(col, row)
  An important property of software *objects* is that they carry around and call functions of their own! They do this using the dot . operator. Here, the image object is calling its own plotPoint function to place a pixel at the given column and row. Functions called in this way are sometimes called *methods*.
- image.saveFile()
  This line creates the new test.png file that holds the png image. It demonstrates another *method* (i.e., function) of the software object named image.

From pixel coordinates to complex coordinates

The problem

Ultimately, we need to plot the Mandelbrot set within the complex plane. However, when we plot points in the image, we must manipulate *pixels* in their own coordinate system.

As the testImage() example shows, pixel coordinates start at (0, 0) (in the lower left) and grow to (width-1, height-1) in the upper right. In the example above, width was 300 and height was 200, giving us a small-ish image that will render quickly.

The Mandelbrot Set, however, lives in the box
-2.0 ≤ x (or real coordinate) ≤ +1.0
and
-1.0 ≤ y (or imaginary coordinate) ≤ +1.0
which is a 3.0 x 2.0 rectangle.

So, we need to convert from each pixel's col integer value to a floating-point value, x. We also need to convert from each pixel's row integer value to the appropriate floating-point value, y.

The solution

One function, named scale, will convert coordinates in general.

So, you'll next write this scale function: scale(pix, pixMax, floatMin,floatMax) that can be run as follows: scale(150, 200, -1.0, 1.0) ==0.5
Here, the arguments mean:

- The first argument is the current pixel value: we are at column 150 or row 150
- The second argument is the maximum possible pixel value: pixels run from 0 to 200 in this case
- The third argument is the minimum floating-point value. *This is what the function will return when the first argument is* 0.
- The fourth argument is the maximum floating-point value. *This is what the function will return when the first argument is* pixMax.

Finally, the return value should be the floating-point value that corresponds to the integer pixel value of the first argument.
The return value will always be somewhere between floatMin and floatMax (inclusive).

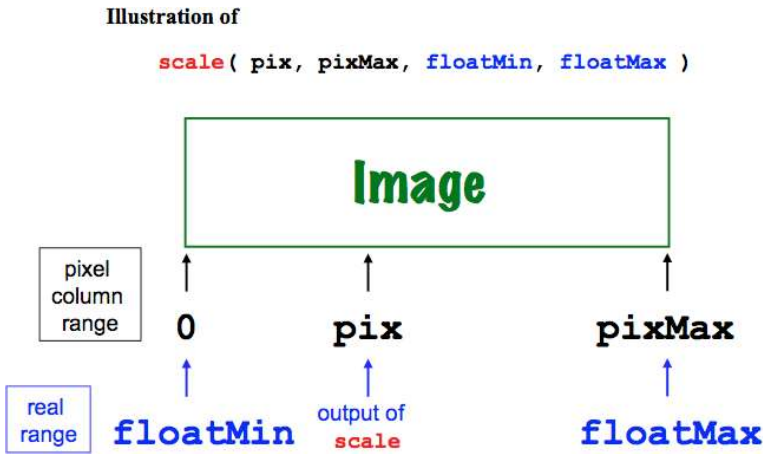
This function will NOT use a loop. In fact, it's just arithmetic. You will need to ask yourself

- How to use the quantity pix / pixMax
- How to use the quantity floatMax – floatMin

A start to the scale function

There is no pixMin because the pixel count always starts at 0.

Again, the idea is that scale will return the floating-point value between floatMin and floatMax that corresponds to the position of the pixel pix, which is somewhere between 0 and pixMax. This diagram illustrates the geometry of these values:



Once you have written your scale function, here are some test cases to try to be sure it is working:

```
In [1]: scale(100,       200,   -2.0,   1.0)    #  halfway from -2 to 1
should be -0.5
Out[1]: -0.5

In [2]: scale(100,       200,   -1.5,   1.5)    #  halfway from -1.5 to 1.5
should be 0.0
Out[2]: 0.0

In [3]: scale(100,       300,   -2.0,   1.0)    #  1/3 of the way from -2 to
1 should be -1.0
Out[3]: -1.0

In [4]: scale(25, 300, -2.0, 1.0)               # 1/12 of the way from -2 to
1 should be -1.75
Out[4]: -1.75

In [5]: scale(299, 300, -2.0, 1.0) # your exact value maydiffer
slightly...
Out[5]: 0.99
```

Note Although we initially described scale as computing x-coordinate (real-axis) floating-point values, your scale function works equally well for both the x- and the y-dimensions. You don't need a separate function for the vertical axis!

Visualizing the Mandelbrot set in black and white: mset

This part asks you to put the pieces from the above sections together into a function named mset(n) that computes the set of points in the Mandelbrot set on the complex plane and creates an image of them, of size width by height.

x and y ranges

To focus on the interesting part of the complex plane, we will limit the ranges of x and y to

-2.0 ≤ x or real coordinate ≤ +1.0
and

-1.0 ≤ y or imaginary coordinate ≤ +1.0

which is a 3.0 x 2.0 rectangle.

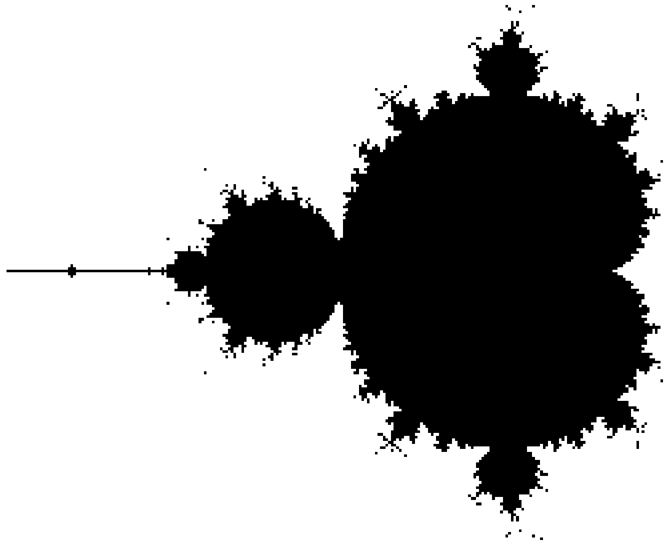
How to get started? Consider this code

```
def mset(n):
    """Creates a 300x200 image of the Mandelbrot set"""
    width = 300
    height = 200
    image = PNGImage(width, height)

    # create a loop in order to draw some pixelsfor col in

    range(width):
        for row in range(height):
            # Use scale twice:
            #     once to create the real part of c (x)
            # x = scale( ..., ..., ..., ...)
            #     once to create the imaginary part of c (y)
            # y = scale( ..., ..., ..., ...)
            # THEN, create c, choose n, and test:
            c = x + y*1j
            if inMSet(c, n):
                image.plotPoint(col, row)

    # we looped through every image pixel; we now write the file
    image.saveFile()
```

To build the Mandelbrot set, you will need to change several behaviors in this function—start where the comment suggests that you should `Use scale twice`:

- For each pixel `col`, you need to compute the *real (x) coordinate* of that pixel in the complex plane. Use the variable `x` to hold this x-coordinate, and use the `scale` function to find it!
- For each pixel `row`, you need to compute the *imaginary (y) coordinate* of that pixel in the complex plane. Use the variable `y` to hold this y-coordinate, and again use the `scale` function to find it! Even though this will be the imaginary *part* of a complex number, it is simply a normal floating-point value.
- Using the real and imaginary parts computed in the prior two steps, create a variable named `c` that holds a ***complex*** value with those real (`x`) and imaginary (`y`) parts, respectively. Recall that you'll need to multiply `y*1j`, not `y*j`!
- Finally, your test for which pixel `col` and `row` values to plot will involve `inMSet`, the third function you wrote. It takes n into your argument

Once you've composed your function, try `mset(25)` and check to be sure thatthe image you get is a black-and-white version of the Mandelbrot set,



e.g., something like this:

Distorted? Rotated? Check to be sure you haven't mixed up `x` and `y`!

Try playing around with different values of `n` to see how the image changes.