

Lab 8: Creating the Mandelbrot Set

Copied from: <https://www.cs.hmc.edu/twiki/bin/view/CS5/Lab8>
on 3/20/2017

Getting started: `execfile` and provided files

Remember you can use `run hw8pr1.py` instead of closing and reopening IPython!

Because you will create images in this lab, you should grab and unzip [hw8pr1.zip](#).

Be sure to keep the `hw8pr1.py` file in the `hw8pr1` folder; it will need the accompanying files.

Submission: You'll submit your `hw8pr1.py` file to the [submission server](#).

The Mandelbrot Set

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

- Use loops and nested loops to solve *complex* problems (quite literally!)
- Develop a program using *incremental design*, i.e., by 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 in your `hw8pr1.py` file:

- Write a function named `mult(c, n)` that returns the product `c` times `n`, but without multiplication. Instead, it should start a value (named `result`) at 0 and repeatedly *add* the value of `c` into that `result`. It should use a `for` loop to make sure that it adds `c` the correct number

of times. After the loop finishes, it should return the result, both conceptually and literally.

The value of `n` will be a positive integer. To get you started, here is a snippet of the function that initializes the value of `result` to 0 and builds the loop itself:

- ```
def mult(c, n):
```
- ```
    """ mult uses only a loop and addition
```
- ```
 to multiply c by the positive integer n
```
- ```
    """
```
- ```
 result = 0
```
- ```
    for i in range(n):
```
- ```
 # update the value of result here in the loop
```

Here are a couple of cases to try:

```
In [1]: mult(6, 7)
Out[1]: 42
```

```
In [2]: mult(1.5, 28)
Out[2]: 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 the assignment statement `z = z**2 + c` for 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 is the `def` line and docstring to get you started:

- ```
def update(c, n):
```
- ```
 """ update starts with z=0 and runs z = z**2 + c
```
- ```
        for a total of n times. It returns the final z.
```
- ```
 """
```

Here are a couple of cases to try:

```
In [1]: update(1, 3)
Out[1]: 5
```

```
In [2]: update(-1, 3)
Out[2]: -1
```

```
In [3]: update(1, 10)
Out[3]: a really big number!
```

```
In [4]: update(-1, 10)
Out[4]: 0
```

You'll use these ideas (through a variant of the `update` function) in building the Mandelbrot Set, next:

## 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:
- $z_{n+1} = z_n^2 + c$

The Mandelbrot set is the collection of all complex numbers  $c$  such that this process does **not** diverge to infinity as  $n$  gets large.

There are other, equivalent definitions of the Mandelbrot set. For example, the Mandelbrot set consists of those points in the complex plane for which the associated *Julia set* is connected. Admittedly, this requires defining Julia sets, which we won't do here... .

The Mandelbrot set is a *fractal*, meaning that its boundary is so complex that it can not be well-approximated by one-dimensional line segments, regardless of how closely one zooms in on it. There are [many available references](#).

## 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 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  $x+yi$ , but in Python the imaginary unit is typed as `1.0j` or `1j`, so that

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

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

Unfortunately, `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.

**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 `TypeError`).

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

## Thinking about the `inMSet` function:

To determine whether or not a number  $c$  is in the Mandelbrot set, you will

- start with  $z_0 = 0 + 0j$  and then
- repeatedly iterate  $z_{n+1} = z_n^2 + c$

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 or not the magnitude of these  $z_k$  go off toward infinity.

Truly determining whether or not 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 good starting point.

The value for infinity can be surprisingly low! 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.

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 step
 Then, it returns
 False as soon as $\text{abs}(z)$ gets larger than 2
 True if $\text{abs}(z)$ 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`.

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

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

```
In [1]: c = 0 + 0j # this one is in the set
```

```
In [2]: inMSet(c, 25)
Out[2]: True
```

```
In [3]: c = 3 + 4j # this one is NOT in the set
WARNING: this one will freeze Python if you're not returning
False
_ as soon as _ the magnitude is larger than 2 (inside the if
or else, inside the loop!)
```

```
In [4]: inMSet(c, 25)
False
```

```
In [5]: c = 0.3 + -0.5j # this one is also in the set
```

```
In [6]: inMSet(c, 25)
Out[6]: True
```

```
In [7]: c = -0.7 + 0.3j # this one is NOT in the set
```

```
In [8]: inMSet(c, 25)
Out[8]: False
```

```
In [9]: c = 0.42 + 0.2j
```

```
In [10]: inMSet(c, 25) # this one _seems_ to be in the set
Out[10]: True
```

```
In [11]: inMSet(c, 50) # but at 50 tries, it turns out that
it's not!
Out[11]: False
```

## Getting too many `True`s?

If so, you might be checking for `abs(z) > 2` *after* the `for` loop finishes. Be sure 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 to get started:

```
from cs5png3 import * # You might already have this line @ the
top...
```

```

def weWantThisPixel(col, row):
 """ a function that returns True if we want
 the pixel at col, row and False otherwise
 """
 if col%10 == 0 and row%10 == 0:
 return True
 else:
 return False

def test():
 """ a function to demonstrate how
 to create and save a png image
 """
 width = 300
 height = 200
 image = PNGImage(width, height)

 # create a loop in order to draw some pixels

 for 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()

```

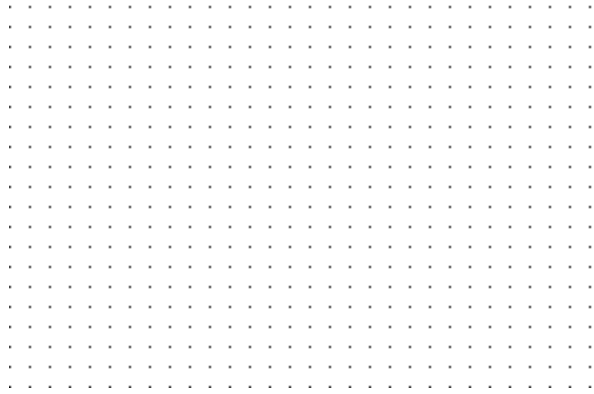
Save this code, and then run it by typing `test()`, with parentheses, at the Python shell.

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 your `hw8pr1.py` file.

Both Windows and Mac computers have nice built-in facilities for looking at png-type images; `png` is short for *portable network graphics*. For many people, double click on the icon of the `test.png` image, and you will see it. (For me, on a Windows machine, opening `test.png` in a browser was even better.)

Either way, 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:





You can zoom in and out of images with the menu options or shortcuts, as well.

## An image thought-experiment to consider...

Before changing the above code, **write a short comment** under the `test` function in your `hw8pr1.py` file describing how the image would change if you changed the line

```
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. On both on Macs and PCs, the image does not have to be re-opened: if you leave the previous image window open, its image will update automatically.

Just for practice, you might try creating other patterns in your image by changing the `test` and `weWantThisPixel` functions appropriately.

## **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 lab 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(pixel, pixelMax, floatMin, floatMax)
```

that can be run as follows:

```
In [1]: scale(150, 200, -1.0, 1.0)
```

Here, the arguments mean the following:

- 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 `pixelMax`.*

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 really just arithmetic. You will need to ask yourself

- How to use the quantity `1.0*pixel / pixelMax`
- How to use the quantity `floatMax - floatMin`

## A start to the `scale` function

To compute this conversion back and forth from pixel coordinates to complex coordinates, write a function that starts as follows:

```
def scale(pixel, pixelMax, floatMin, floatMax):
 """ scale accepts
 pixel, the CURRENT pixel column (or row)
```

```
pixMax, the total # of pixel columns
floatMin, the min floating-point value
floatMax, the max floating-point value
scale returns the floating-point value that
corresponds to pix
"""
```

The docstring describes the arguments:

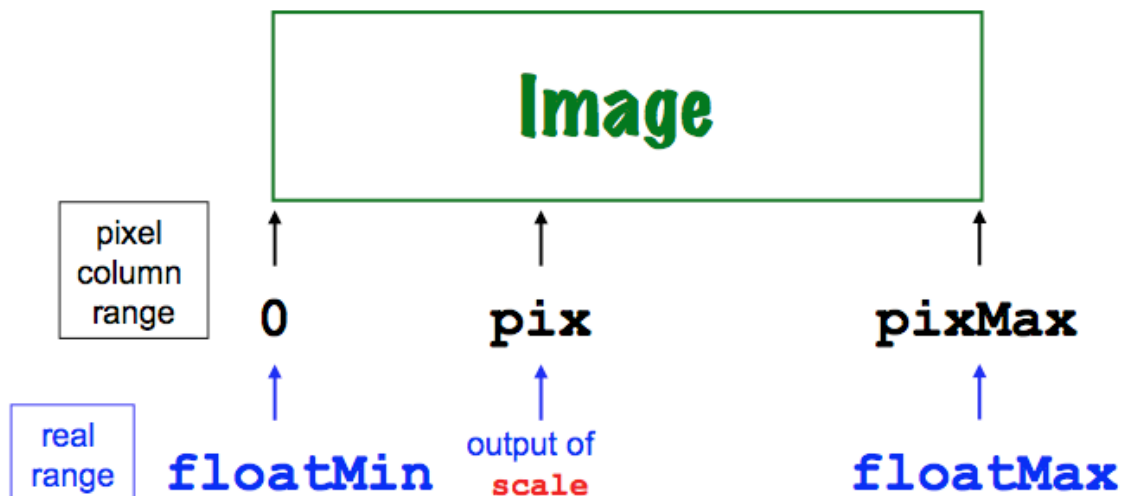
- `pix`, an integer representing a pixel column
- `pixMax`, the total number of pixel columns available
- `floatMin`, the floating-point lower endpoint of the image's axis
- `floatMax`, the floating-point upper endpoint of the image's axis

Note that 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:

**Illustration of**

```
scale(pix, pixMax, floatMin, floatMax)
```



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 may
differ 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()` that computes the set of points in the Mandelbrot set on the complex plane and creates a bitmap of them, of size `width` by `height`. To focus on the interesting part of the complex plane, we will limit the ranges of `x` and `y` to

$-2.0 \leq x$  or real coordinate  $\leq +1.0$

and

$-1.0 \leq y$  or imaginary coordinate  $\leq +1.0$

which is a 3.0 x 2.0 rectangle.

**How to get started?** Start by copying the code from the `test` function and renaming it as `mset`:

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

```

create a loop in order to draw some pixels

for col in range(width):
 for row in range(height):
 # here is where you will need
 # to create the complex number, c!
 # Use scale twice:
 # once to create the real part of c (x)
 # once to create the imag. part of c (y)
 # THEN, test if it's in the M. Set:
 if inMSet(c, n):
 image.plotPoint(col, row)

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

```

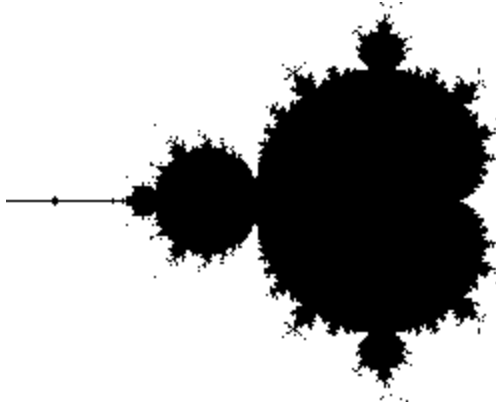
To build the Mandelbrot set, you will need to change a number of behaviors in this function—start where the comment suggests that *here is where...*:

- 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 first function you wrote. You'll want to specify a value for the argument named `n` to that `inMSet` function. I'd start with a value of 25 for `n`.

Once you've composed your function, try

```
In [1]: mset()
```

and check to be sure that the image you get is a black-and-white version of the Mandelbrot set, e.g., something like this:



## Adding features to your `mset` function

### No magic numbers!

*Magic Numbers* are simply literal numeric values that you have typed into your code. They're called *magic* numbers because if someone tries to read your code, the values and purpose of those numbers seem to have been pulled out of a hat... For example, your `mset` function might call `inMSet(c, 25)` (at least, mine did). A newcomer to your code (and this problem) would have no idea what that `25` represented or why you chose that value.

To keep your code as flexible and expandable as possible, it's a good idea to avoid using these "magic numbers" for important quantities in various places in your functions. Instead, it's better to collect all of those magic numbers at the very top of your functions (after the docstring) and to give them useful names that suggest their purpose. It's common, but not required, to use all caps for these values—for example, you might have the line

```
NUMITER = 25 # of updates
```

where `NUMITER` is the number of iterations to be used by the `inMSet` function. The function call would then look like

```
inMSet(c, NUMITER)
```

In addition to being clearer, this makes it *much* easier to add or change functionality in your code—all of the important quantities are defined **one time** and in **one place**.

For this part of the lab, **move all of your magic numbers to the top of the function and give them descriptive names**. For example, these five lines are a good starting point:

```
NUMITER = 25 # of updates, from above
XMIN = -2.0 # the smallest real coordinate value
XMAX = 1.0 # the largest real coordinate value
YMIN = -1.0 # the smallest imag coordinate value
YMAX = 1.0 # the largest imag coordinate value
```

These variables can then be changed in one place in order to alter the "window" to the Mandelbrot Set being plotted.

In the next part, you'll change these values to zoom into the Mandelbrot set.

## Zooming in!

---

For this part, simply run your `mset` function with some different values of `XMAX`, `XMIN`, `YMAX`, and `YMIN` to create images of different parts of the Mandelbrot Set.

For example, try the values

```
NUMITER = 25
XMIN = -1.3
XMAX = -1.0
YMIN = .1
YMAX = .3
```

Then, try this range again, this time using `NUMITER = 50` - you'll see that fewer points are included because of the additional time to escape... !

Another range worth checking out is the "infinite snowmen":

```
NUMITER = 25
XMIN = -1.2
XMAX = -.6
YMIN = -.5
YMAX = -.1
```

**Optionally**, feel free to look around to find another set of values that shows an interesting piece of the set—and note what they are in a comment in your `hw8pr1.py` file. As a guide, you might consider the suggestions at [this M. set "atlas"](#) or [this page on the "Seahorse Valley"](#).

Note that the *aspect ratio* of the image is 3:2 (horizontal:vertical), and if you keep this aspect ratio in your ranges, the set will be scaled naturally.



It will work with different ratios, but to maintain the natural scaling, you would have to change the height and width of the image accordingly. Or you could *compute* the height and width, but that's not required for this lab.

## Changing colors...

---

You don't have to use black and white!

The `image.plotPoint` method accepts an optional third argument that represents the *color* of the point you'd like to plot. Here is an example:

```
image.plotPoint(col, row, (0,0,255))
```

The third argument here is a list that uses *parentheses* instead of square-brackets. Parenthesized lists are called *tuples* in Python. Tuples are faster to access than lists, but their elements cannot be assigned to, so they're often used for constants such as colors.

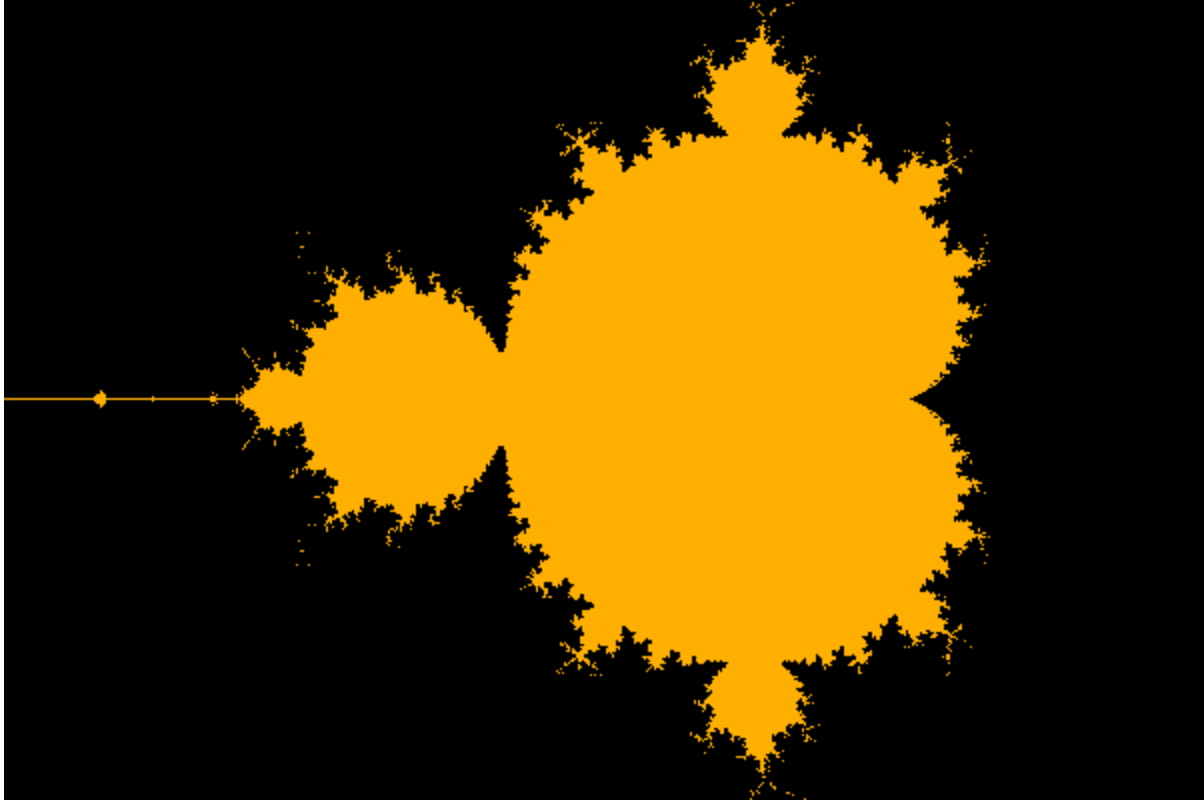
The three elements of the color tuple above are *red*, then *green*, then *blue*: each of those three color components must be an integer from 0 to 255. Thus, the tuple above, `(0,0,255)` is pure blue.

To change the *background* of the set, add the lines

```
else:
 image.plotPoint(col, row, (0,0,0))
```

*within* the loops that run over each `col` and each `row`. This will explicitly plot, in black, all of the points *not* in the Mandelbrot Set

**Try it out!** perhaps first by changing your Mandelbrot set to orange (255,175,0) on top of a black background (0,0,0), a combination particularly suitable for this week of the year...:



Then, change the colors to something more to your liking!

Feel free to enlarge your image, too—it will take longer to render, but you'll be able to resolve more detail in the result!

You've completed Lab #8 and the `hw8pr1.py` problem on this week's homework—be sure to submit that!

If you'd like to visualize the different *escape velocities* or change the resolution or perhaps *mandelbrotify* another image, these next sections describe how to add features to your Mandelbrot-set program. They're optional, but fun!

### ***Completely Optional: Visualizing escape velocities***

---

These extensions are optional (but fun!)—this first one lets you see the relative speeds with which the points in and/or around the Mandelbrot set escape to infinity.

Images of fractals often use color to represent *how fast* points are diverging toward infinity when they are not contained in the fractal itself. For this problem, create a new version of `mset` called `msetColor`. Simply copy-and-paste your old code, because its basic behavior will be the same. However, you should alter the `msetColor` function so that it plots points that escape to infinity *more quickly* in different colors.

Thus, have your `msetColor` function plot the Mandelbrot set as before. In addition, however, use at least three different colors to show how quickly points outside the set are diverging—this is their "escape velocity." Making this change will require a change to your `inMSet` helper function, as well. We suggest that you copy, paste, and rename `inMSet` so that you can change the new version without affecting the old one.

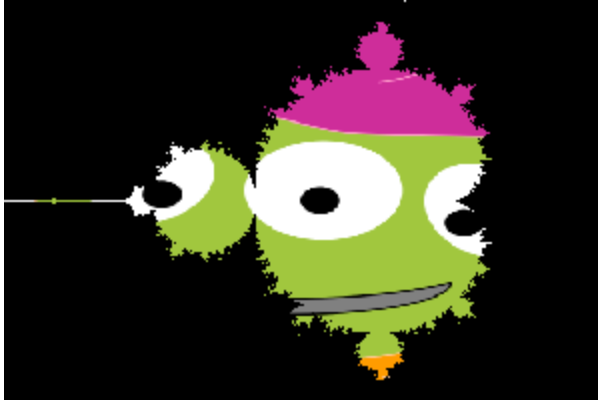
There are several ways to measure the "escape velocity" of a particular point. One is to look at the resulting magnitude after the iterative updates. Another is to count the number of iterations required before it "escapes" to a magnitude that is greater than 2. An advantage of the latter approach is that there are fewer different escape velocities to deal with.

Choose one of those approaches—or design another of your own—to implement `msetColor`.

## **Mandelbrotifying another image**

The `png` library can read images, too. (As a warning, it's not overly fast in converting them to Python lists...)

The result is that you can use another image's pixels to determine the look of the points in (or out) of your Mandelbrot set, e.g.,



Here is an example function to show how to read in an image (the `alien.png` image from the `pngs` folder:

```
def example():
 """ shows how to access the pixels of an image
 inputPixels is a list of rows, each of which is a list
of columns,
 each of which is a list [r,g,b]
 """

 inputPixels = getRGB("./pngs/alien.png")
 inputPixels = inputPixels[::-1] # the rows are reversed

 height = len(inputPixels)
 width = len(inputPixels[0])
 image = PNGImage(width, height)

 for col in range(width):
 for row in range(height):
 if (col%10 < 5) and (row%10 < 5): # only plot some
of the pixels
 image.plotPoint(col, row, inputPixels[row][col])

 image.saveFile()
```

The above code produces the following image:



Try *Mandebrotifying* this image—or another `png` you'd like to use...

When you are done—and it can be surprisingly addicting—you are done with lab!

Be sure to submit your `hw8pr1.py` file....