# Creating the Mandelbrot Set

## Getting started: Provided files

Because you will create images in this lab, you should grab and unzip **mandelbrot.zip**. Be sure to keep the `mandelbrot.py` file in the `mandelbrot` folder: it will need the accompanying files. The standard IDLE will work fine -- all of this week's graphics are still images.

## 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 file:

- Write a function named `mult( c, n )` that returns the product of 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. Here is a snippet of the function that initializes the value of `result` to 0 and builds the loop itself, in order to get you started:

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

  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 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:

```
>>> update( 1, 3 )

5

>>> update( -1, 3 )

-1

>>> update( 1, 10 )

a really big number!

>>> update( -1, 10 )

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. Choose a complex number $c$ . With this $c$ in mind, start with

$z_0 = 0$

and 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. Of course, this requires defining Julia sets... We will leave such details aside for now.

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. In fact, the Mandelbrot set's boundary has a dimension of 2 (!), though the details of this are better left to the many available references.

## The `inMSet` function

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

```
>>> c = 3 + 4j
```

```
>>> c
(3+4j)

>>> abs(c)
5.0

>>> c**2
(-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 do not have an ordering defined on them since they are essentially points in a 2D space). So you cannot do something like `c > 2`. However, you CAN compare the magnitude (i.e., the absolute value) of a complex number to the number 2, e.g., `abs(c) > 2`.

## Back to 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 repeatedly iterate $z_{n+1} = z_n^2 + c$ to see if this sequence of $z_0$, $z_1$, $z_2$, ... stays bounded. That is, we would need to know whether or not the magnitude of these $z_k$ go off toward infinity.

*Really* determining whether or not this sequence goes off to infinity would take forever! To check this computationally, 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"

The first value above is `n`. That is, `n` represents the number of times we are willing to run the *z*-updating process. This is the second input to the function `inMSet(c, n)`. This is a value you will want to experiment with, but 25 is a reasonable initial value.

The second value above, the one that represents infinity, can be surprisingly low! One can prove, though we won't, that if the absolute value of the complex number $z$ ever gets larger than `2`during that 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.

## Writing `inMSet`

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

**In this case**, it's better to copy and adapt that old function. *Don't call `update` directly.*

Thus, the first lines of `inMSet` will look like this:

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

As the docstring notes, 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`) and then update the value of that variable within a loop, just as you did with `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**:

```
>>> c = 0 + 0j         # this one is in the set

>>> inMSet(c, 25)

True



>>> 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... !

>>> inMSet(c, 25)

False
```

```
>>> c = 0.3 + -0.5j    # this one is also in the set

>>> inMSet(c, 25)

True


>>> c = -0.7 + 0.3j    # this one is NOT in the set

>>> inMSet(c, 25)

False


>>> c = 0.42 + 0.2j

>>> inMSet(c, 25)      # this one seems to be in the set

True


>>> inMSet(c, 50)      # but it turns out that it's not!

False
```

**Getting too many `Trues`?**

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

There is a subtle reason this won't work. 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

This file uses *different* approach to graphics (writing out images) than previous homework problems. It will work on SnowLeopard, Windows, and pretty much any OS you might have without any special handling. ☺

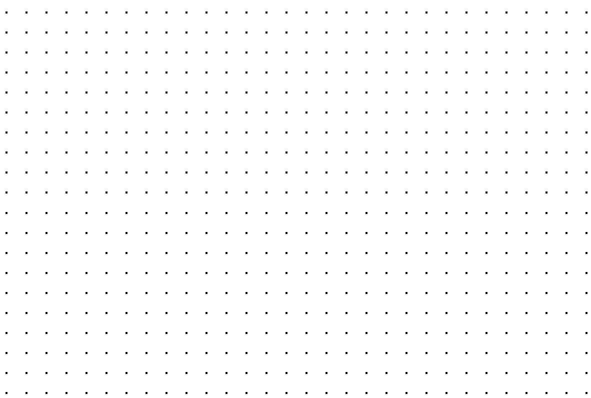
Here is a bit of code to get you started - try it out!

```
from cs5png import *    # You may already have this line...

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 ) == True:
                image.plotPoint(col, row)

    # we looped through every image pixel; we now write the file

    image.saveFile()
```

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

If everything goes well it will run through the nested loops and print a message that the file `test.png` has been created. It should be 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*). Simply double click on the icon of the `test.png` image, and you will see it. 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 bitmaps with the built-in buttons on Windows; on Macs, similar commands are available via menu and keyboard shortcuts.

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

Before changing the above code, **write a short comment or triple-quoted string** 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. It seems that both on Macs and PCs, the image does not have to be re-opened: if you leave the previous preview 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, so 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 can 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 in order to, well, plot a point at the given column and row. Functions called in this way are sometimes called *methods*.
- `image.saveFile()` This line actually 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

Ultimately, we are trying to plot the Mandelbrot set within a complex coordinate system. However, when we plot points in the image, we must manipulate *pixels*.

As the `testImage()` example shows, pixel values always start at (0, 0) (in the lower left) and grow to (width-1, height-1) in the upper right. In the example above `width` and `height` were both 200, giving us a reasonably sized image.

However, the Mandelbrot Set 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`.

Thus, we will write a function named scale:

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

that can be run as follows:

```
>>> scale( 150, 200, -1.0, 1.0 )
```

Here, the inputs mean the following:

- the first input is the current pixel value: we are at col 150 or row 150
- the second input is the maximum possible pixel value: pixels run from 0 to 200 in this case
- the third input is the minimum floating-point value. *This is what the function will output when the input is* `0`.
- the fourth input is the maximum floating-point value. *This is what the function will output when the input is* `pixelMax`.

Finally, the ***output*** should be the floating-point value that corresponds to the integer pixel value of the first input. The output 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*pix / pixMax`
- How to use the quantity `floatMax - floatMin`

## Writing the `scale` function

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

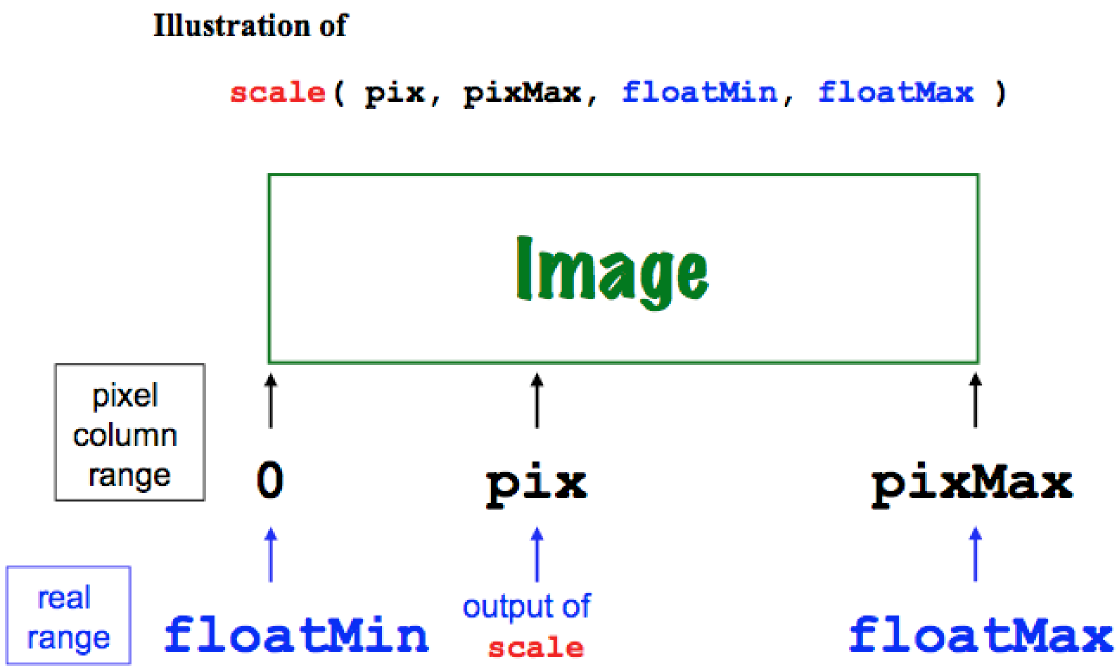
```
def scale(pix, pixMax, floatMin, floatMax):
    """ scale takes in
            pix, the CURRENT pixel column (or row)
            pixMax, the total # of pixel columns
            floatMin, the min floating-point value
            floatMax, the max floating-point value
        scale returns the floating-point value that
            corresponds to pix
    """
```

The docstring describes the inputs:

- `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 real axis (x-axis)
- `floatMax`, the floating-point upper endpoint of the image's real axis (x-axis).

Note that there is no `pixMin` because the pixel count always starts at 0.

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:

```
>>> scale(100, 200, -2.0, 1.0)  # halfway from -2 to 1

-0.5

>>> scale(100, 200, -1.5, 1.5)  # halfway from -1.5 to 1.5

0.0

>>> scale(100, 300, -2.0, 1.0)  # 1/3 of the way from -2 to 1

-1.0

>>> scale(25, 300, -2.0, 1.0)

-1.75

>>> scale(299, 300, -2.0, 1.0)  # your exact value may differ slightly

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(width, height)`

which will generate images of width `width` and height `height` with that computes the set of points in the Mandelbrot set on the complex plane and creates a bitmap of them. We will use images and, to focus on the interesting part of the complex plane, we will limit the ranges within the complex plane 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?** 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!
            if inMSet( c, n ) == True:
                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 input named `n` to that `inMSet` function. I'd start with a value of `25` for `n`.

Once you've composed your function, try

```
>>> 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: