

TARIQ RASHID

# PYTHON: MAKE YOUR OWN MANDELBROT SET

With a smattering of Python and a bit of clever maths you too can create a bit of beautiful chaos inside your Linux box.

## WHY DO THIS?

- Create hugely complex results from a simple equation.
- Turn this into pretty graphs!

The following images are closeups of parts of this same fractal. The Mandelbrot set is in fact infinitely detailed, and contains a wide variety of patterns.

The image below depicts the famous Mandelbrot set. You may have seen it on posters, music videos or demonstrations of computer power – back in the 1990s it took hours to plot it using home computers. Despite its organic intricate appearance it is in fact the result of extremely simple mathematics.

The maths hasn't changed since the 1990s, but computing power has come on in leaps and bounds, so the psychedelic beauty of the Mandelbrot set can be ours to play with. We'll be using the Python programming language because it's popular, easy to learn, and has well established numerical and plotting extensions. It's no accident that Python is the tool of choice in the scientific community, as well as powering some of the world's largest infrastructures.

As an interpreted, rather than compiled language, Python gives immediate feedback. We're going to take this interactivity one step further by using IPython, which is Python packaged with an interactive web interface. You can just type instructions and get results straight away. You avoid having to edit source

code files and using shells to execute the programs. You can also share your web-based "notebooks" by exporting them, or sharing them online.

There are pre-packaged distributions of IPython including the numerical and graphical extensions, so you don't have to worry about installing and configuring the right versions and their many dependencies. Even better, because IPython is used purely through a web browser, you can use one of several online services offering IPython in the cloud. You don't have to install any software at all, and you can work on your code and demonstrate the results from any internet-connected device that supports a modern browser.

The code in this article series was tested with the online service from [wakari.io](http://wakari.io), which offers free trial accounts, and the locally installed IPython Anaconda free distribution from [continuum.io](http://continuum.io).

Whether you select an online service or install your own IPython distribution, check to see if it works by firing it up and clicking on New Notebook. Type **2\*3** into it and click the Run button, which looks like an audio play selector. If it responds with '6', great – you have a working IPython environment!

## Beginning Python

Let's now learn just enough Python to make our own Mandelbrot fractal. Fire up IPython and open a new notebook. In the next ready cell, labelled **In []**, type the following code and click Play (or press Ctrl+Enter).

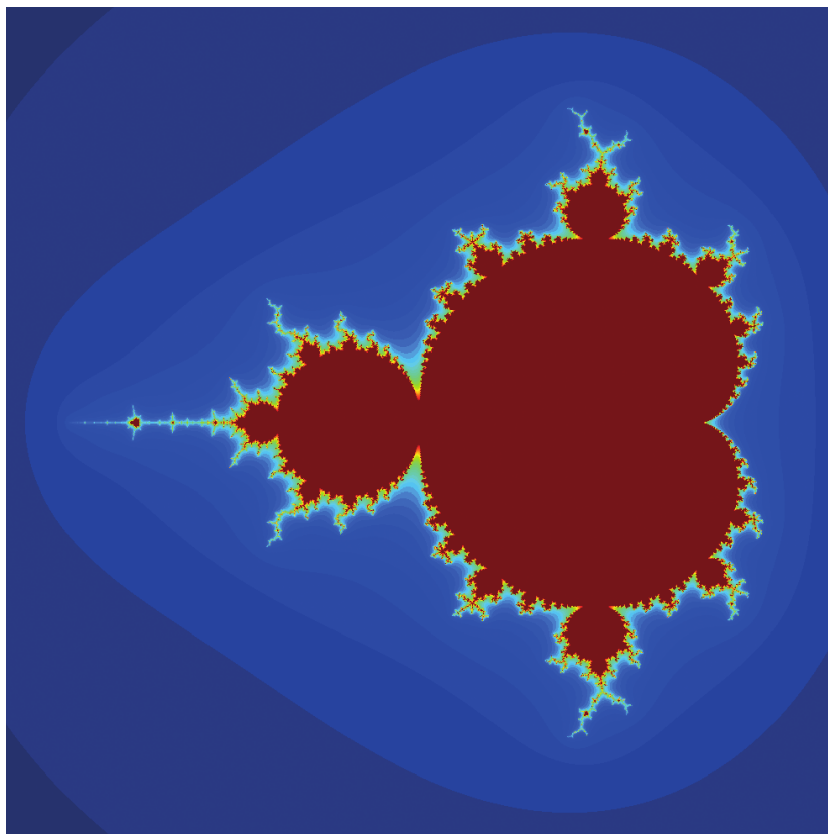
```
print "Hello World!"
```

You should get a response that simply prints the phrase "Hello World!" You should see that issuing the second instruction didn't remove the previous cell with its instruction and output answer. This is useful when you're slowly building up a solution of several parts.

The following code introduces the key idea of variables. Enter and run it in a new cell. If there is no new empty cell, click the button with the downward pointing arrow labelled 'Insert Cell Below', not to be confused with the one labelled "Move Cell Down".

```
x = 10
print x
print x+5
print z
```

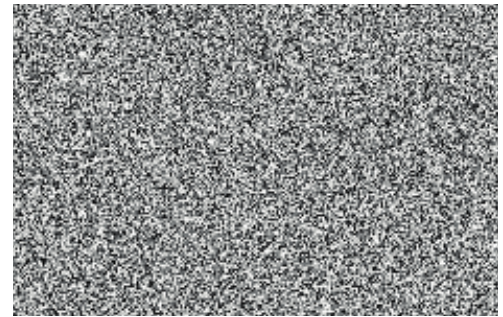
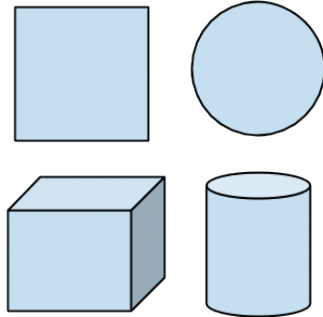
The first line, **x = 10**, looks like a mathematical statement that says x is 10. In Python this means that x is set to 10, that is, the value 10 is placed in a virtual box called x. That 10 stays there until further



## Fractals

Before we dive in and start coding, let's take a step back and consider again some of the patterns we find in nature or create ourselves. Look at the following three images.

The first set of shapes is very regular. They're what most people would consider mathematical shapes, but there isn't enough interesting detail in them to hold our attention for long. The last image



is entirely random noise – there isn't a lack of detail, but now there isn't enough structure in the image to keep us interested. The cauliflower has repeated patterns at different scales. These patterns also have just the right amount of variation to keep us interested. This is a common theme throughout nature, where clouds, mountains, rivers, trees, blood vessels etc all have self-similar

patterns with just the right amount of unpredictable variation. These patterns are called fractals.

The Mandelbrot fractals appear natural, organic and even beautiful because they too sit at that fine line between order and chaos, having structures that have self-similar patterns but infused with just enough variation – and parts of these fractals do look like plants, lightning or natural coastlines.

notice. We shouldn't be surprised that **print x** prints the value of `x`, which is 10. Why doesn't it just print `x`? Python will evaluate whatever it can, and `x` can be evaluated to the value 10, so it prints that. The next line, **print x+5** evaluates `x+5`, which is `10+5` or 15, so we expect it to print 15.

What happens with the line **print z** when we haven't assigned a value to it like we have with `x`? We get an error message telling us about the error of our ways, trying to be helpful as possible so we can fix it.

### Automating lots of work

Computers are great for doing similar tasks many times – they don't mind and they're very quick compared with humans with calculators. Let's see if we can get a computer to print the first 10 squared numbers, starting with 0 squared, 1 squared, then 2 squared and so on. We expect to see a printout of something like 0, 1, 4, 9, 16, 25, and so on. Issue the following code into the next ready cell and run it.

```
range(10)
```

You should get a list of 10 numbers, from 0 up to 9. This is great because we got the computer to do the work to create the list – we didn't have to do it ourselves.

A common way to get computers to do things repeatedly is by using loops. The word loop does give you the right impression of something going round and round potentially endlessly. Rather than define a loop, it's easiest to see a simple one. Enter and run following code in a new cell.

```
for n in range(10):
```

```
    print n
```

```
    pass
```

```
print "done"
```

There are three new things going on here. The first line has the **range(10)** command that we saw before, which creates a list of numbers from 0 to 9. The **for n**

**in** is the bit that creates a loop, and here it does something for every number in the list, and keeps count by assigning the current value to the variable **n**. We saw variables earlier, and this is just like assigning **n=0** during the first pass of the loop, then **n=1**, then **n=2**, until **n=9**, the last item in the list.

The next line **print n** prints the value of `n`, just as before. We expect all the numbers in the list to be printed. But notice the indent before **print n**. This is important in Python as indents are used meaningfully to show which instructions are subservient to others, in this case, the loop created by **for n in ...**. The loop ends when the code stops being indented. Here, we've used a **pass** instruction to highlight the end of the loop (**pass** is superfluous, and Python will ignore it but it helps the interpreter exit a code block. You can remove it if you want). This means we only expect **done** to be printed once, and not 10 times.

It should be clear now that we can print the squares by printing **n\*n**. In fact we can make the output more helpful with phrases like "The square of 3 is 9". The following code shows this change inside the loop. Note how the variables are not inside quotes and are therefore evaluated.

```
for n in range(10):
```

```
    print "The square of", n, "is", n*n
```

```
    pass
```

```
print "done"
```

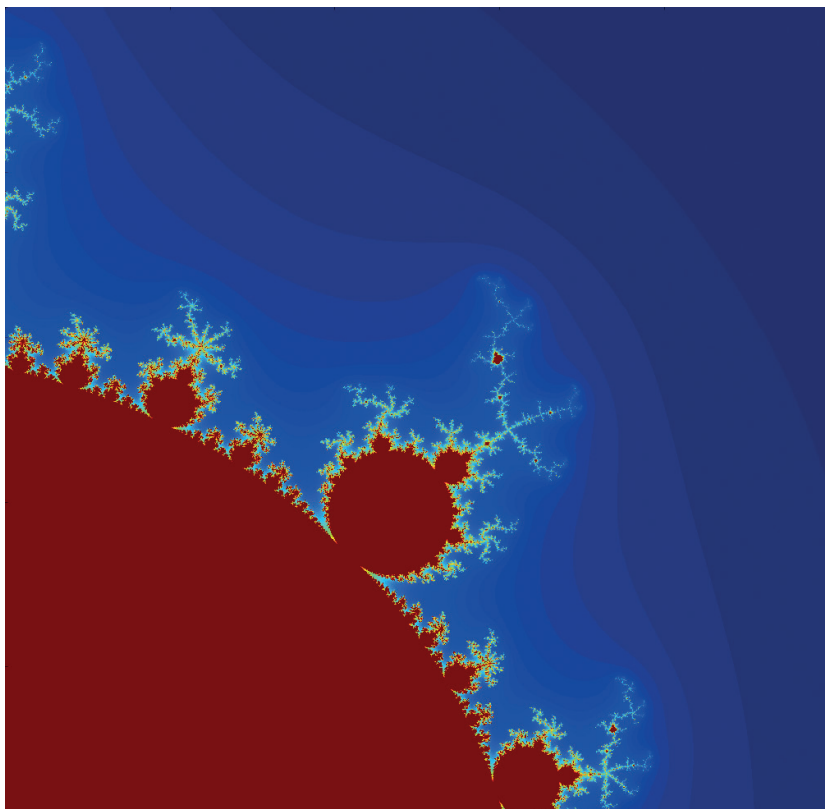
This is already quite powerful. We can get the computer to do a lot of work very quickly. We could easily make the number of loop iterations much larger by using **range(100)** or even **range(100000)** if we wanted. Try it!

### Functions

Python makes it easy to create reusable computer instructions. Like mathematical functions, these reusable snippets of code, also called functions, stand

#### LV PRO TIP

Explore the Mandelbrot fractal using the interactive XaoS open source software at <http://bit.ly/1vLdz52> or through a web browser <http://bit.ly/1lbXYL1>.



The amazing thing is that all this detail and variety emerges from the behaviour of a very simple mathematical function;  $z^2+c$ .

on their own if you define them sufficiently well, and allow you to write shorter more elegant code. Why shorter code? Because invoking a function by its name many times is better than writing out all the function code many times. (By sufficiently well defined we mean being clear about what kinds of input a function expects, and what kind of output it produces. Some functions will only take numbers as input, so you can't supply it with text.)

Let's look at a simple function and play with it. Enter the following code and run it.

```
# function that takes 2 numbers as input
```

```
# and outputs their average
```

```
def avg(x,y):
```

```
    print "first input is", x
```

```
    print "second input is", y
```

```
    a = (x + y) / 2.0
```

```
    print "average is", a
```

```
    return a
```

The first two lines beginning with **#** are ignored by Python, but we use them to add comments for future readers. The next bit, **def avg(x,y)**, tells Python we are about to define a new reusable function. That's the **def** keyword. The **avg** part is the name we've given it. It could have been called "banana" or "pluto" but it makes sense to use names that remind us what the function actually does. The bits in brackets **(x,y)** tells Python that this function takes two inputs, to be called **x** and **y** inside the definition of the function.

Now that we've signalled to Python that we're about to define a function, we need to actually tell it what the function is to do. This definition of the function is indented under **def**. The first and second numbers, **x** and **y**, which the function receives when it is invoked are printed. The next bit calculates  $(x+y)/2.0$  and assigns the value to the variable named **a**, before it is printed. The last statement says **return a**. This is the end of the function and tells Python what to return back to whoever invoked it.

When we ran this code, it didn't seem to do anything. There were no numbers produced. That's because all we've done is define the function; we haven't used it yet. What has actually happened is that Python has noted this function and will keep it ready for when we want to use it.

In the next cell enter **avg(2,4)** to invoke this function with the inputs 2 and 4. The output should be what we expect, with the function printing a statement about the two input values and the average it calculated. The following shows the function definition and the results of calling it with **avg(2,4)** and also bigger values (200,

### LV PRO TIP

In Python, like many programming languages, the **=** equals sign means "is set to". So **x=5** means **x** is set to 5 until, and if, it is updated later. This is different from the mathematical or logical equivalence, which can confuse new programmers.

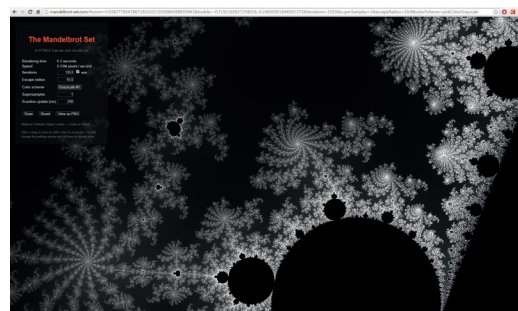
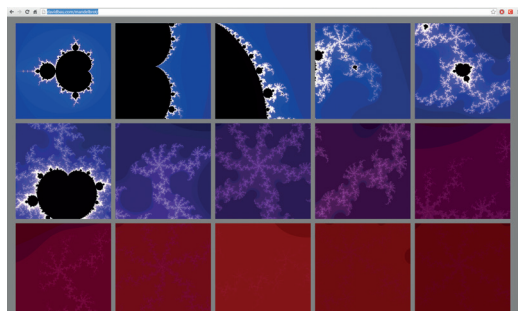
### Explore fractals in a web browser

You can explore the Mandelbrot fractal without installing any software at all. The following two are implemented entirely in JavaScript, so you only need a modern browser to explore the fractals. Have a go!

The explorer at <http://davidbau.com/mandelbrot> enables you to progressively click on points to create a new closer

image. You can then point at a new point in the closer view, or go back and pick a new point to explore from.

The explorer at <http://mandelbrot-set.com> requires you to use your pointer to select a rectangle to zoom into. The level of detail calculated rapidly is impressive. You can select different colour schemes with the options menu.



The simple folk of the 90s would be awed and terrified to see the Mandelbrot set calculated in a browser.

301). Experiment with your own inputs.

You may have noticed that the function code which calculates the average divides the sum of the two inputs by "2.0" and not just "2". This is because "2" is an integer and so Python will round the result to an integer as well, which we don't want here.

## Arrays

Arrays are just tables of values. You refer to particular cells with the row and column number, just like you would with cells in a spreadsheet.

Enter and run the following code.

```
a = zeros( [3,2] )
print a
```

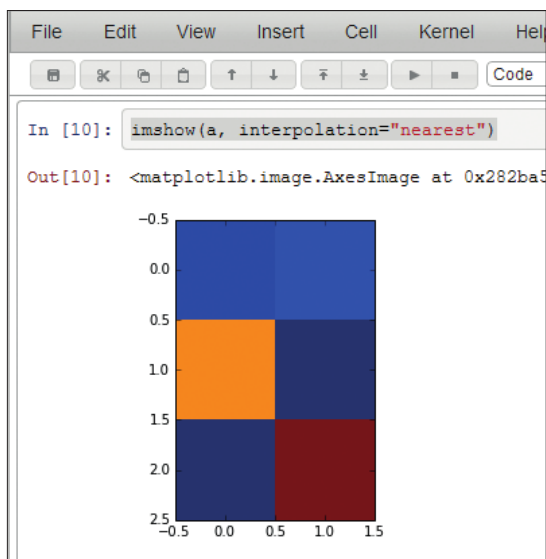
This creates an array of shape 3 by 2, with all the cells set to the value zero and assigns the whole thing to a variable named **a**. Printing **a** shows the array full of zeros in what looks like a table with three rows and two columns.

Now let's modify the contents of this array. The following code shows how you can refer to specific cells to overwrite them with new values. It's just like referring to spreadsheet cells or street map grid references.

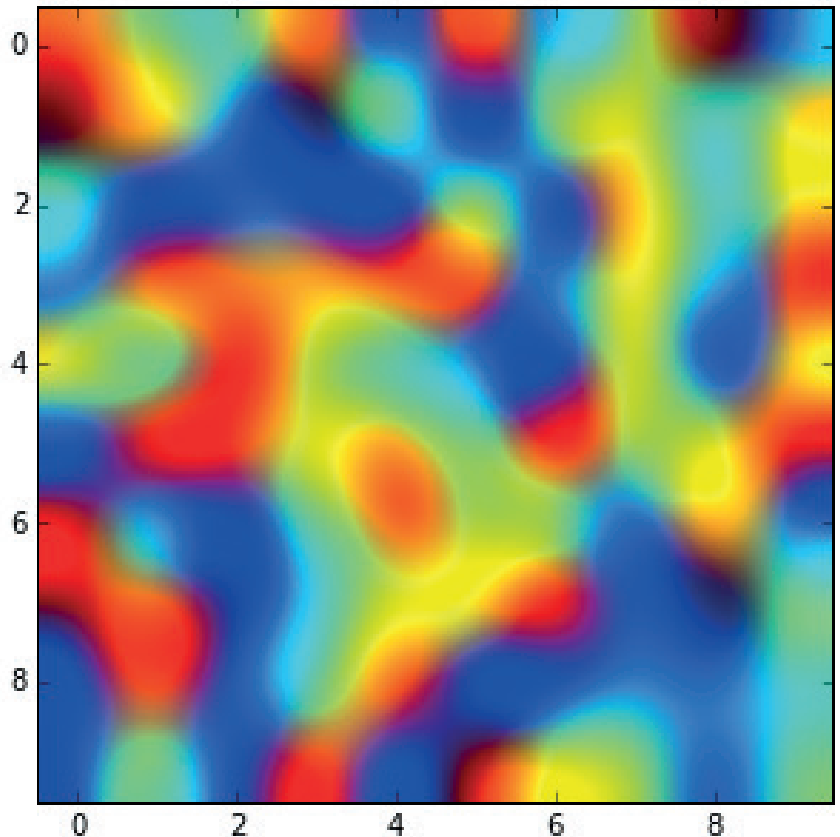
```
a[0,0] = 1
a[0,1] = 2
a[1,0] = 9
a[2,1] = 12
print a
```

The first line updates the cell at row zero and column zero with the value 1, overwriting whatever was there before. The other lines are similar updates, with a final printout.

Now that we know how to set the value of cells in an array, how do we look them up without printing out the entire array? We've been doing it already. We simply use the expressions like **a[1,2]** or **a[2,1]** to refer to the content of these cells. The code shows us



The array cells which have the same value also have the same colour. When we plot the Mandelbrot set, we'll be using this very same **imshow** instruction.



An example of the images produced by the code teaser, below. Can you figure out how it works?

doing just this.

```
print a[0,1]
v = a[1,0]
print v
```

Remember that the column and row numbering starts from 0 and not 1, so the top-left is at [0,0] not [1,1]. This also means that the bottom-right is at [2,1] not [3,2].

## Plotting arrays

Visualising arrays graphically is sometimes more insightful than looking at large tables of numbers. We can think of them as flat two-dimensional surfaces, coloured according to the value at each cell in the array. Let's plot the small 3 x 2 array we created above:

```
imshow(a, interpolation="nearest")
```

The instruction to create a plot is **imshow()**, and the first parameter is the array we want to plot. That last bit, **interpolation**, is there to tell Python not to try to blend the colours to make the plot look smoother, which it does by default. The output is shown in the image below.

As a teaser of things to come, the following short code will plot different images every time you run it. See if you can work out how it works using the online Python documentation.

```
import numpy as np
figsize(5,5)
imshow(np.random.rand(10,10), interpolation="lanczos")
```

To really appreciate the Mandelbrot fractal we need to experience for ourselves the unexpected behaviour of very simple mathematical functions that lead to

### LV PRO TIP

Remember that in Python, indents have meaning. Instructions that are subservient to another are indented. This includes function definitions and the contents of loops. An errant space or tab can cause hard-to-spot errors.

### LV PRO TIP

Experiment with the **imshow()** function by trying different options explained in its documentation at <http://bit.ly/1lu1mkB>.

its intricate patterns. You don't need to be an expert in mathematics to follow and appreciate the same surprise felt by those researchers who first pictured the Mandelbrot set in the late 1970s.

### Iteration

You can imagine a mathematical function as a machine that does some work. Its job is to take numbers in one end, the input, and spit out a new number out the other end, the output. What happens if we feed the output of one of these functions back into it again as the input?

Let's try it with the function "divide by 3" and starting value of 9. We get the sequence 9, 3, 1,  $\frac{1}{3}$ , ... You can see the numbers getting ever smaller, and in fact they'd never get to zero.

Iteration simply means doing the same thing again and again to produce a series of outputs. If the values keep growing larger forever, like those from the "multiply by 2" function, we say the values diverge. If they approach a finite value, like zero, like the "divide by 3" function, we say the values converge.

### Starting conditions

Some functions behave differently depending on their starting value – in these cases, we say they are sensitive to initial conditions. Consider the function "square it" which simply takes an input and multiplies it by itself. If the starting value is greater than 1, the successive outputs keep growing larger. If the seed value is smaller than 1, the values keep getting smaller. A seed value of exactly 1 stays the same, and separates the two domains of divergence and convergence. This idea of domains of different behaviour is important for the Mandelbrot fractal because that is exactly what it is showing – regions of divergence and convergence.

Other kinds of behaviour were discovered only recently in the long history of mathematics. The Logistic Map is a function  $rx(1-x)$  that was developed by scientists trying to model population growth. The behaviour is very sensitive to the starting value and

parameter  $r$ .

For some seed values and parameters  $r$ , the function behaves like a divergent or a convergent function. But for some, such as  $x=0.2$  and  $r=3$ , the function seems to oscillate. This is certainly unexpected behaviour!

What's more, for some other values like  $x=0.2$  and  $r=4$  the behaviour breaks down into something unrecognisable, apparently random. This is known as chaos. Not only is this behaviour surprising from such a simple function, it was only really appreciated in the last 100 years or so of mathematics, which itself goes back thousands of years.

We're going to see if we can feed our functions a new kind of number called a complex number. You may remember these from school, but if not, we'll explain them here.

Complex numbers have two parts. They are two-dimensional, and so can be used to refer to points in a flat plane just like grid coordinates. These two parts just happen to be called the real and imaginary parts. We can add and subtract these numbers very easily by combining the real and imaginary parts separately. For example  $1+2i$  added to  $3+4i$  is  $4+6i$  (4 is the real part and  $6i$  is the imaginary part).

Multiplying complex numbers is just like school algebra. The brackets are expanded and similar terms are recombined, but there is one special rule for complex numbers: any  $i^2$  is replaced by  $-1$ . For example  $(2+3i)(1+4i)$  is  $2 + 8i + i^3 + 12i^2$  which simplifies to  $(-10+11i)$ .

Python can work with complex numbers out of the box. We use the form **complex(a,b)** to tell Python we mean **a+ib** where **a** is the real part and **b** is the imaginary part of the complex number. Try the following in an IPython notebook.

```
# assign the complex number (2+3i) to c
```

```
c = complex(2,3)
```

```
print c
```

```
# print c multiplied by (1 - 4i)
```

```
print c * complex(1,-4)
```

```
# print c squared
```

```
print c*c
```

The behaviours we saw earlier, including oscillation and chaos, also occur for functions working on complex numbers, except they take place in two dimensions. The successive values from a function are called orbits. The plot below-left shows successive values for the simple function  $z^2+c$  with  $z$  starting at **0** and  $c=0.4+0.4i$ . You can see that after an initial circling, the values suddenly diverge – not the behaviour we were used to at school with less interesting functions!

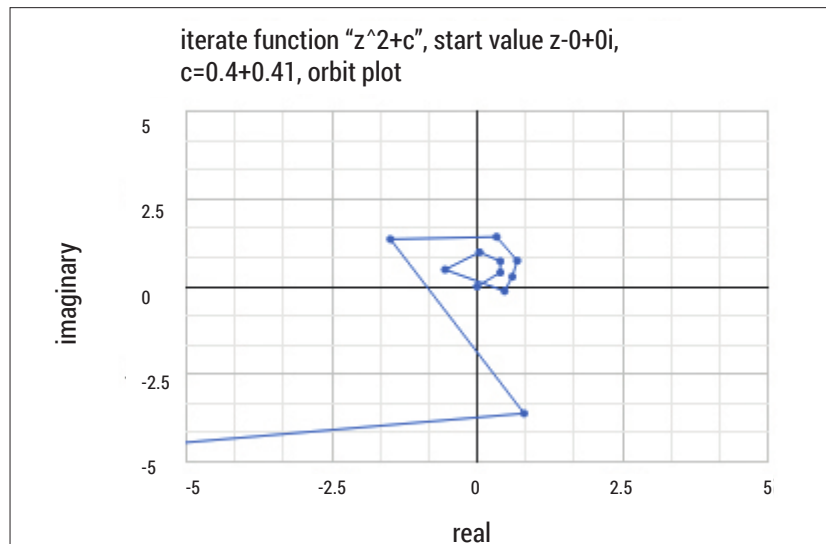
### Mandelbrot fractal recipe

The Mandelbrot set is just a two-dimensional atlas coloured to show which regions of its world, the two-dimensional complex plane, diverge or converge

#### LV PRO TIP

It's good practice to design your functions to always return the same result for the same set of inputs. When your programs grow in complexity, this is a guarantee that will help keep your code understandable and aid debugging.

Orbit plot showing successive values of  $z^2+c$  when  $c=0.4+0.4i$ . You can see an initial circling then a sudden divergence.



when the amazingly simple function  $z^2+c$  is applied iteratively. Here  $z$  starts at  $0$ , and  $c$  is the complex number representing the chosen point on the complex plane. If a chosen point diverges and gets bigger and bigger it is considered outside the Mandelbrot set. We could colour all such points one colour, but it is common to chose a colour according to how rapid the divergence is. The points inside the set, those which don't diverge, are usually coloured black.

It turns out, to many people's surprise, that the boundary between the two is not a boring shape like a circle but is incredibly detailed and intricate, and in fact beautiful – it is the famous Mandelbrot fractal!

## Mandelbrot set in Python

We'll build up a Python program to calculate and plot a Mandelbrot fractal in easy to understand pieces.

Let's start at the core of the Mandelbrot calculation. For each point being tested on a selected rectangle of the complex plane, the function  $z^2+c$  is repeatedly iterated the resulting values may diverge rapidly. The point is coloured according to how quickly that function diverges. This means the Python function we want to write returns the number of iterations it takes to diverge beyond a threshold magnitude. The function still needs to calculate successive values of  $z^2+c$ , it just doesn't have to return them. So the start and end of the core calculating Python function, which we'll call `mandel()`, looks like the following.

```
def mandel(c):
```

```
..
```

```
..
```

```
return iterations
```

What if the point doesn't diverge? We can define the maximum number of iterations the function is to be applied before giving up. We can pass it as a parameter to the core `mandel()` function. The function would then look something like `mandel(c, maxiter)`. Why would we need to change it? Well, as you explore the Mandelbrot fractal's finer detail, you need more iterations to establish whether very close points behave differently or not.

We now have a `mandel()` function that takes the test point  $c$ , and the maximum number of iterations as its parameters:

```
def mandel(c, maxiter):
```

```
z = complex(0,0)
```

```
for iteration in xrange(maxiter):
```

```
...
```

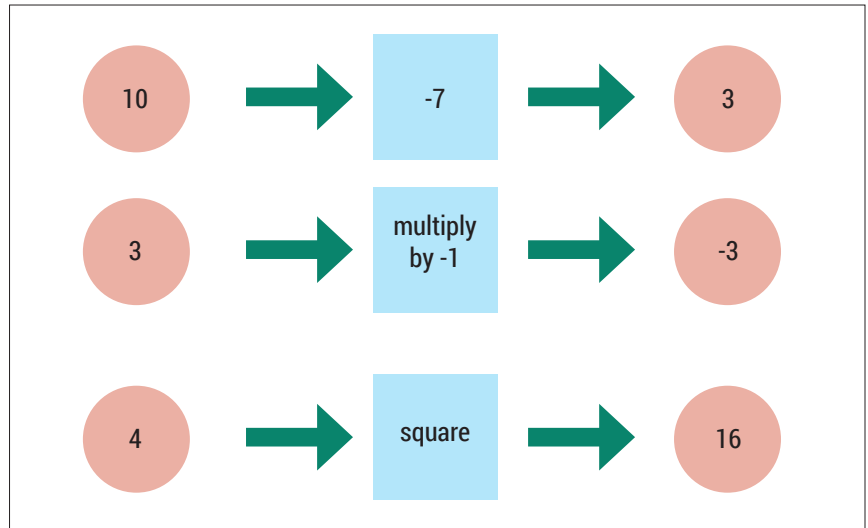
```
...
```

```
...
```

```
pass
```

```
return iteration
```

We set the starting value of  $z$  to be zero, or more precisely  $(0+0i)$ . The `for ..` code loop, which iterates a maximum of `maxiter` times, keeps count in the variable named `iteration`. The end of the function is still returning the iteration count, whether that reaches



the maximum `maxiter`, or is stopped sooner by a magnitude threshold test. What's left is to fill in the code describing the iterated function  $z^2+c$  and then check to see if the threshold has been breached. These are fairly easy, so let's write them out and explain them.

```
def mandel(c, maxiter):
```

```
z = complex(0,0)
```

```
for iteration in xrange(maxiter):
```

```
z = (z*z) + c
```

```
if abs(z) > 4:
```

```
break
```

```
pass
```

```
pass
```

```
return iteration
```

Here we've added the  $z = (z * z) + c$  instructions, which calculates the next value of  $z$  based on the current value and the chosen  $c$ . We then check to see if the magnitude, or absolute value denoted `abs()` in Python, of  $c$  is greater than 4, and if it is, the instruction `break` simply breaks out of the `for` loop. Once this happens there are no more instructions after the loop, and so the `mandel(c,maxiter)` function returns the

Some examples of functions: the `square` functions takes an input of 4 and squares it, producing the resulting answer 16.

### LV PRO TIP

There are several kinds of convergence you'll find if you experiment with these and similar functions. Successive values will get closer to, but never reach, zero or a finite value. Alternatively, they will fluctuate above and below a value as they get ever closer to it. In some cases they might orbit about a finite number of different attractors, that is, multiple points which seem to pull them closer.

## Resources

If you'd like a fuller explanation of the mathematics and Python discussed in this series, you'll find the Make Your Own Mandelbrot ebook (Amazon and Google) takes a slower journey with more examples and discussion. All the source code can be found at the blog.

- Source code <http://makeyourownmandelbrot.blogspot.co.uk/2014/04/sharing-code.html>

- IPython <http://ipython.org/install.html>

- Wakari.io [www.wakari.io](http://www.wakari.io)

- XaoS <http://matek.hu/xaos/doku.php>

- Blog <http://makeyourownmandelbrot.blogspot.co.uk>

- Amazon Kindle ebook [www.amazon.co.uk/dp/B00JFIEC2A](http://www.amazon.co.uk/dp/B00JFIEC2A)

### LV PRO TIP

Complex numbers really aren't complex. The term is an accident of history, and sadly puts some people off them. If they were called composite numbers, that would reflect the fact that they are made up of two independent parts.

### PRO TIP

You can find a visual representation and gentler explanation of the algorithm to calculate and plot the Mandelbrot fractal at <http://bit.ly/1qk78e3>

## Arithmetic with complex numbers

The following table summarises how to add, subtract and multiply complex numbers.

Operation	How to do it
Add the two complex numbers $(a + bi)$ and $(c + di)$	$(a + bi) + (c + di) = (a+c) + (b+d)i$ Add the real and imaginary parts independently.
Subtract the complex number $(c + di)$ from $(a + bi)$	$(a + bi) - (c + di) = (a-c) + (b-d)i$ Subtract the real and imaginary parts independently.
Multiply the two complex numbers $(a + bi) * (c + di)$	$(a + bi) * (c + di) = (ac + adi + bci + dbi^2) = (ac-bd) + (ad+bc)i$ Expand out the terms and apply the special rule that $i^2$ is $-1$ . Then collect real and imaginary parts to make a neat answer.

value of iteration. If the point doesn't diverge, then  $\text{abs}(z)$  is never more than 4, so the **for** loop simply keeps running until the count reaches the maximum iterations, and it is this maximum that is then finally returned by the **mandel(c, maxiter)** function.

### The atlas

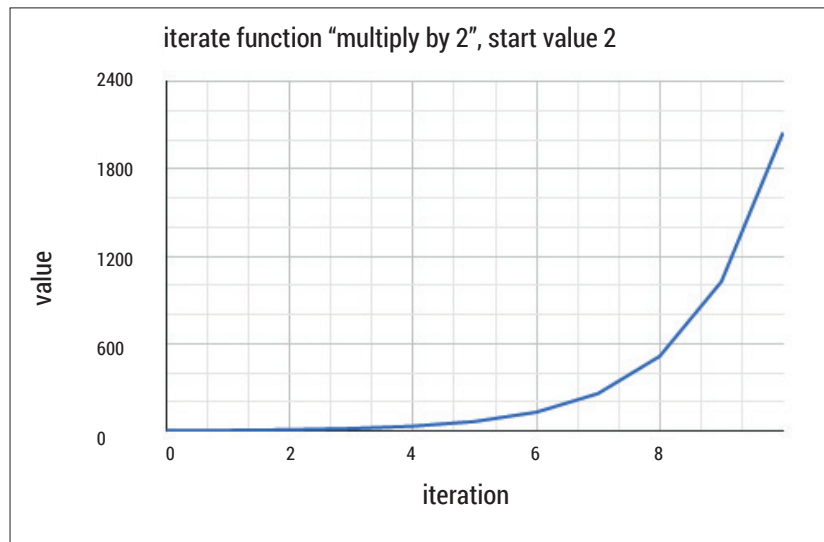
We now need to define in Python which part of this

complex plane we are interested in. We also need to divide up this section into regularly spaced points, ultimately representing pixels in an image.

**"We want to colour each region according to its convergence or divergence behaviour."**

Python has a function **linspace()** which divides an interval into evenly spaced points. Let's imagine we want a rectangle with bottom-left at **(-2,-2)** and the top-right at **(4,2)**. This has a horizontal length of 6, and a vertical height of 4. Let's divide the horizontal length into 12 sections, and the vertical into 8. The following Python code shows how you can use the familiar Python loops over each element of **linspace** lists to create the coordinates for each test point within this rectangle.

After each iteration of the 'multiply by 2' function, the results get ever further from each other – this is divergence.



```
x_list = linspace(-2.0, 4.0, 13)
```

```
y_list = linspace(-2.0, 2.0, 9)
```

```
for x in x_list:
```

```
    for y in y_list:
```

```
        print x,y
```

```
    pass
```

```
pass
```

Next we need to find a way of associating these points with the pixels in an array of colour values that could be plotted using the **imshow()** function we used earlier. The complex plane region is just a list of points, represented by complex numbers, and these don't have a colour associated with them to plot. We need to give the **imshow()** plotting function something that contains colour information. Also, **imshow()** expects to plot a two-dimensional array where the contents of a cell represent the colour to be plotted, not a long list of complex numbers like the ones we created earlier.

Given that the rows and columns of the plotted array need to increment in whole units, we can simply place each of the evenly spaced points between the bottom-left and top-right into the array. So if the points were 0.5 units apart on the complex plane, they would be 1 unit apart in the array.

Let's now define the complex plane region. Enter and run the following code. It makes sense to place this at the top of your *IPython* notebook because it sets out up front which region you are interested in. Use the button marked as 'Insert Cell Above' to create a new cell at the top.

```
# set the location and size of the complex plane rectangle
```

```
xvalues = linspace(-2.25, 0.75, 1000)
```

```
yvalues = linspace(-1.5, 1.5, 1000)
```

```
# size of these lists of x and y values
```

```
xlen = len(xvalues)
```

```
ylen = len(yvalues)
```

The first instruction creates a list of 1000 points evenly placed between -2.25 and 0.75, inclusive. These will be the horizontal divisions of the rectangle, and we'll call the list **xvalues**. Similarly, **yvalues** is the list of 1000 evenly spaced points between -1.5 and 1.5. The last two lines simply take the length of the lists and assign them to variables.

The following code creates the image array of colour values of size **xlen** by **ylen**. We've called it **atlas** because we want to colour each region according to its convergence or divergence behaviour.

```
atlas = empty((xlen, ylen))
```

We're almost there! All that remains is to fill this array with colour values and plot it using **imshow()**. The following code uses loops to fill it with the returned values from the **mandel()** function.

```
for ix in xrange(xlen):
```

```
    for iy in xrange(ylen):
```

```
        cx = xvalues[ix]
```

```
        cy = yvalues[iy]
```

```
        c = complex(cx, cy)
```

```
        atlas[ix, iy] = mandel(c, 40)
```

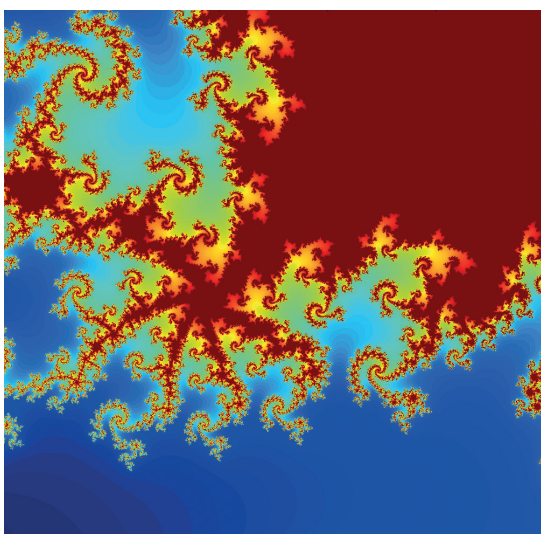
```
    pass
```

```
pass
```

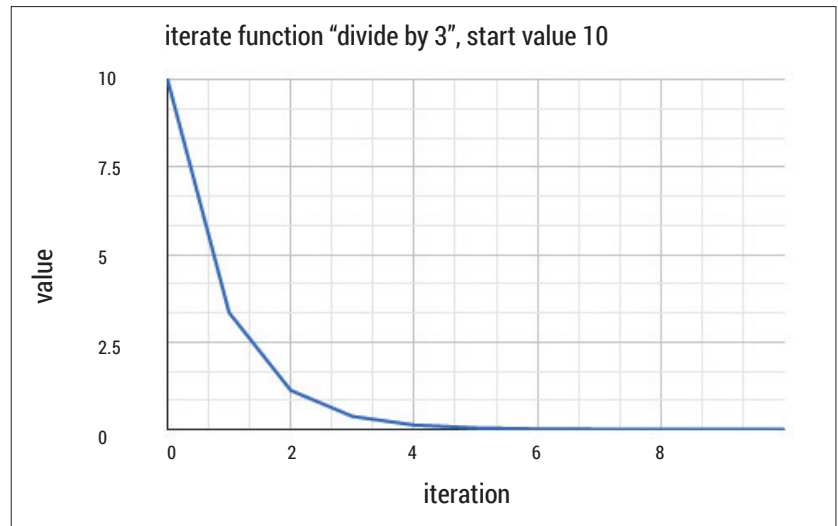
You'll recognise that this code is simply two loops, one inside the other. The loops count through the rows and columns of the **atlas** array using variables **ix** and **iy**. These counts refer to the contents of the array, which are also counted from **0** to **xlen-1**, and not **1** to **xlen**. You may have noticed that we use **xrange** instead of **range**. **range** would work, but for very large lists **xrange** is more efficient because it doesn't actually create a list, but gives you the contents as you ask for them.

These two loops enable us to refer to every cell of the array using **atlas[ix, iy]**. The code inside the loops uses the counts **ix** and **iy** to look up the actual complex number to be tested by the **mandel()** function. The real and imaginary parts were in the **xvalues** and **yvalues** lists we created earlier, and can be dug out using **xvalues[ix]** and **yvalues[iy]**.

The last part inside the loops is updating the contents of the array with the return value from the **mandel()** function.



It looks different from the image on the first page of this tutorial, but this is actually just a magnified section.



In the divide by three function, the result trends towards, but never reaches, zero.

That's it, the hard work is done! Now let's see the results. In a new cell, enter the following code.

```
figsize(18,18)
```

```
imshow(atlas.T, interpolation="nearest")
```

The first line sets the size of the plot to 18 by 18 because the default is too small. The **imshow** instruction plots the array. We also refer to **atlas** with a **.T** appended to it because otherwise the array is plotted on its side compared to what we want to see.

Run the code and you'll see the Mandelbrot set.

You can zoom into parts of the Mandelbrot set by changing the bottom-left and top-right points of the complex plane region. We simply change the code that sets the **xvalues** and **yvalues**. For example, using the rectangle from earlier in this guide with the values **(-0.22 - 0.70i)** bottom-left and **(-0.21 - 0.69i)** as top-right means setting the following **xvalues** and **yvalues** as follows:

```
# set the location and size of the atlas rectangle
```

```
xvalues = linspace(-0.22, -0.21, 1000)
```


```
yvalues = linspace(-0.70, -0.69, 1000)
```

The resulting image was quite undefined because we set too low a value for maximum iterations. Change it from 40 to 120 as follows:

```
atlas[ix, iy] = mandel(c, 120)
```

The result is a more detailed image, as shown below-left. It's really quite beautiful!

The complete Python code we've built up to plot our own Mandelbrot fractals is available for you to look over at <http://makeyourownmandelbrot.blogspot.co.uk/2014/04/sharing-code.html>. I've added comments to help remind you what each code section does.

Next month we'll look at the Julia fractals, which are intimately related to the Mandelbrot fractals. They're even more beautiful in my opinion! We'll also extend our 2D fractals into interactive alien 3D landscapes you'll be able to explore. 

#### LV PRO TIP

The code for the full programs to calculate and plot the Mandelbrot and Julia fractals is online at <http://bit.ly/1qpnlSE>.

#### LV PRO TIP

Don't forget you can explore the Mandelbrot and other fractals using the interactive XaoS open source software at <http://bit.ly/1vLdz52> or through a web browser <http://bit.ly/1lbXYL1>.

Tariq spends his time grappling with enterprise IT problems, informed by two decades of working with open technology.