Assignment 4 (python + numpy/C) (30/40 + 5 points)

All solutions should be stored in a directory called `assignment4` in your private repository.

Mathematical background (0 points)

In this assignment, you will make a program which plots a fractal called the Mandelbrot set. For an explanation of how to draw it, read this section or check out `https://plus.maths.org/content/unveiling-mandelbrot-set` We will use $\mathbb{C}$ to refer to the complex numbers. If you are unfamiliar with complex numbers, think of them as the normal plane of all points (a, b) with an operation called complex multiplication $(a, b) \cdot_{\mathbb{C}} (c, d) = (ac - bd, ad + bc)$.

For each complex number c, define a function[1] $f_c(z) = z^2 + c$. For some numbers c, the sequence $0, f_c(0), f_c(f_c(0)), f_c(f_c(f_c(0))), \ldots$ will grow towards infinity. For others, it will not. The *Mandelbrot set* is the set of all c for which this sequence does `not` tend towards infinity. It is known (and you may assume) that if at any point the sequence is above 2, it tends towards infinity (and if not, it does not). So for example, you can check that (1, 0) is not in the Mandelbrot set, but (-1, 0) is.

To draw a picture of the Mandelbrot set, then, you can color each complex number c as black if it is in the Mandelbrot set, and otherwise color it according to how many times you need to apply f_c to 0 to get above 2. In practice, the nicest way to do this is to choose some fine, regularly spaced grid of points in the complex plane, each corresponding to a pixel in the image you are drawing, and for each point compute elements of the sequence $0, f_c(0), f_c(f_c(0)), \ldots$ until you either go above 2, in which case you note how many steps it took, or until you reach some preset threshold (say 1000) steps, in which case you conclude it's probably in the Mandelbrot set. Then, use some color scale mapping numbers between 0 and 1000 to colors and color the pixels accordingly.

4.1: Python implementation (6 points)

First, create a Python script which chooses some rectangle in the complex plane, colors that rectangle according to the method above, and saves the coloring as an image. You do not have to make a user interface for your script yet, but it should be easy to change the area drawn by modifying parameters inside the script.
You *are* allowed to use numpy for this part of the problem - see clarifications.
Name of file: `mandelbrot_1.py`

[1] In coordinates, if $z = (x, y)$ and $c = (a, b)$, $f_c(z) = (x^2 - y^2 + a, 2xy + b)$

4.2: numpy implementation (6 points)

Make a similar script to 4.1 so that all computationally heavy bits use numpy arrays. Compare the runtime on some input of your choosing. How much is gained by switching to numpy?
Name of file: `mandelbrot_2.py`, `report2.txt`

4.3: Integrated C implementation (10 points)

Choose your favorite way of integrating C with Python. (We recommend cython.) Make yet another script using that to do the same computations, and compare the runtime to your previous examples. How much is gained by switching to C?
Name of file: `mandelbrot_3.py`, `report3.txt`

4.4: An alternative integrated C implementation (INF4331 only: 10 points)

Redo 4.3 using SWIG. (If you used that in 4.3, use cython instead.) Compare the runtime as before, but also comment on which implementation technique you preferred, and why. Which was easier to write? Which seems easier to change?
Name of files: `mandelbrot_4.py`, `report4.txt`

4.5: User interface (4 points)

Provide a command line user interface for your script. The design of this is up to you, but it should provide instructions for its by calling it with a `--help` flag, and it should be possible to specify which region of the plane to draw, the resolution to draw in and output image filename. It should also be possible to switch between your 3 implementations with a command line argument.

4.6: Packaging and unit tests (4 points)

Make your implementation into a Python module or package and make a setup script. Your package should include a method `compute_mandelbrot(xmin, xmax, ymin, ymax, Nx, Ny, max_escape_time=1000, plot_filename=None)` which returns an $Nx \times Ny$ array of the escape times of evenly sampled points in the rectangle [xmin, xmax] $\times$ [ymin, ymax]. If `plot_filename` is supplied, an image should be rendered and saved to the specified location. You are free to make other methods which you think add useful functionality.

Further, make two unit tests. One should test that if you choose a region of the plane which is entirely outside the Mandelbrot set after 0 iterations (like

the rectangle [3, 4] $\times$ [3, 4]), your script notices. Another should check that if you take a region which is entirely inside the Mandelbrot set, your script should not say they are outside. You are free to choose which unit testing framework to use for writing your tests, provided it is reasonably standard. py.test is a good option.
Name of test file: `test_mandelbrot.py`

4.7: More color scales + art contest (3 bonus points)

Use your script to make pretty pictures. Add support for additional color scales to your script, giving the user at least 3 visually distinct options, and experiment with different color scales and locations to render, and make up a picture you think look cool. The picture deemed nicest will win a small surprise. (The contest is in addition to the bonus points.)
Name of your contest submission: `contest_image.jpg/png/whatever`

4.8: Self-replication (2 bonus points)

Make a short (but non-empty!) python script which prints its own source code. Your script should *not* accomplish this task by accessing the file containing its own source code, so it should *not* open any files for reading or call system utilities which do.

`replicator.py`

Clarifications

- Previously, the mathematical background contained a mistake. It has now been corrected, and it now (c Mandelbrot set is the set of c for which the sequence does *not* diverge to infinity.
- Prevously, 4.3 required use of weave, which is not ported to Python 3. Now, it requires Cython, which is ported.
- In 4.1, you are allowed to use numpy to store your numbers etc., but try to make your computationally intensive bits use python loops instead of numpy array computations. To give an example, in 4.1, if `a` is a numpy array, you could do things like

  ```
  for i in a.shape[0]:
      b[i] = 3 * a[i]
  ```

  whereas in 4.2, replace that with the array computation

  ```
  b = 3*a
  ```

  The point of doing both 4.1 and 4.2 is to see what a difference using numpy to *compute* makes, so to iterate, it's *fine* if you use numpy to store your numbers in 4.1.
- Using external libraries like matplotlib, PIL to create images is fine, as long as your script does the Mandelbrot computations.
- In 4.8, some people have expressed concern that they do not understand the requirements. The "spirit" of the requirements is that your script should print exactly the same thing as is contained in the source code, and this should not be accomplished by actually reading from the file containing its source code. The source code should also not be an empty file. :)

  As a rough heuristic which may or may not be helpful, your script is fine if it would still work even if a magical goblin deleted the file containing its source code between it being passed to the Python interpreter, but before it actually executes your program.