

COSC122 (2012) Lab 4

Recursion

Quiz Due Date: 5pm Friday, 10 August
Quiz Drop Dead Date: 5pm Monday, 13 August

Goals

This lab will give you some practice with several problems that can be solved using *recursion*, and a number of recursive algorithms. In this lab you will:

- compute a number of simple problems (factorials, exponentiation, *etc.*) using recursive functions;
- have more fun with linked lists (or at least improve your understanding of linked lists);
- compare iterative and recursive solutions and think about where recursive methods are (in)appropriate;
- experiment with, and draw *fractals*.

You should be familiar with the material in Chapter 4 (1st edition Chapter 3) of the textbook before attempting this lab.

The `recursion.py` module has a number of functions and function stubs for solving simple problems recursively.

Fibonacci Numbers

A *Fibonacci number* is a number that features in the Fibonacci sequence of numbers:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Given that the first two numbers are 1 and 1 (the *1st* and *2nd* numbers, respectively), all other numbers in the sequence are calculated as the sum of the last two numbers. For example, the 5th Fibonacci number is the 3rd + the 4th ($2 + 3 = 5$).

```
def fib_iterative(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_n_minus2 = 0
        fib_n_minus1 = 1 #start with fib of 2 which equals 1
        for i in range(2,n+1): #there will be n+1 numbers including the 0'th
            curr_fib = fib_n_minus1+fib_n_minus2
            fib_n_minus2 = fib_n_minus1
            fib_n_minus1 = curr_fib
        return curr_fib
```

Try calculating various Fibonacci numbers with `fib_iterative`, eg, 5, 10, 100, 1000, 2000, 10000. You will see that the fibonacci numbers become very large. Luckily Python can handle such large integers but we might need to think about how much space such numbers will take up.

It is far more *elegant* to solve this problem recursively. Think about solving $fib(4)$ as ' $fib(3) + fib(2)$ ', and solving $fib(3)$ as ' $fib(2) + fib(1)$ ', and so on... —that is, solving $fib(n)$ is the same as ' $fib(n-1) + fib(n-2)$ '. The recursive version has been implemented for you in the `fib_recursive` function in the `recursion.py` module.

While running the method `fib_recursive`, try to inspect how the Python stack behaves (as explained in the lectures).

Try calculating the following Fibonacci numbers using `fib_recursive`: 5, 10, 20, 30, 40. You will find that although recursion is elegant, in this case it isn't so efficient. ¹

To see why the recursive method gets out of hand, try drawing the start of the tree of calls from `fib_recursive(10)` - don't go all the way down to `fib_recursive(0)` though. For example, `fib_recursive(10)` calls `fib_recursive(9)` and `fib_recursive(8)`, then `fib_recursive(9)` calls `fib_recursive(8)` and `fib_recursive(7)`, etc...

Examine the other provided functions (`power`, `gcd`, `linked_list_print`, and `str_length`).

> **Complete question 1 in Lab Quiz 4.**

Recursive Implementations

Your task is to implement the remaining functions in the `recursion` module that are missing implementations:

- `linked_list_length(list_node)`: Returns the length of the linked list with the head at the given node.
- `linked_list_reverse_print(list_node)`: Reverse prints out the linked list with the head at the given node.
- `in_list(list_node, item)`: Returns True if the item is found in the list.
- `recursive_string_print(s)`: Prints characters one per line.
- `factorial`: Returns the factorial of n
- `quick_power`: Returns x^n .

The function recursive function definition for `quick_power` is given below:

$$x^n = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ (x^{n/2})^2 & \text{if } n \text{ is even} \\ x \cdot x^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

When thinking about how to write these functions, you need to think of a *base case* where the method does not recurse any further, and each time it calls itself, it must move towards the base case.

Think about string slicing in Python and how this would make the recursive print very inefficient in terms of space used on the stack. Why will the recursive print for a linked list be a lot more efficient in this regard?

Using recursion to traverse linked list is more natural than iteration (eg, the while loops that were used in the linked list lab). Compare your recursive print function with the print functions for printing stacks and queues in the linked list lab. Good compilers/interpreters will usually turn such simple tail recursion in to iteration for efficiency so we get the elegance of recursion with the efficiency of iteration. Unfortunately Python doesn't do well in this area, so be careful to only use recursion where it isn't likely to cause stack blow-outs.

> **Complete questions 2 to 6 in Lab Quiz 4.**

¹In fact, to calculate the n^{th} fibonacci number, the number additions will be roughly equal to the fibonacci number being calculated - and this grows very fast, ie, the function will be $O(2^n)$ - think about why. The iterative version can be used to see how many additions the recursive version would be doing for large n 's.

Fractals

A *fractal* is a geometrical shape that possesses an infinite level of detail. They are generated through a repeating pattern, such that zooming into them reveals smaller versions of the larger whole. One of the most popular examples of this is the *Mandelbrot set*, shown in Figure 1; zooming into the Mandelbrot set reveals an incredible level of geometric complexity and eventually, smaller versions of the entire set again.

While fractals may *seem* very complicated, their final form after many iterations simply masks the simple drawing instructions that they are built on.

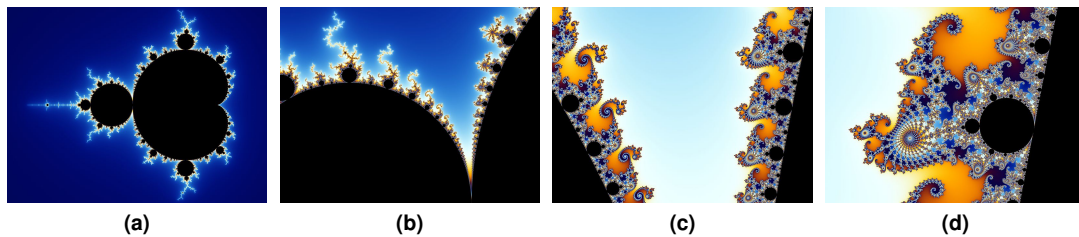


Figure 1: Zooming into the Mandelbrot set; from http://en.wikipedia.org/wiki/Mandelbrot_set.

Turtle Graphics

The fractals we're going to be looking at can be drawn using a technique known as *turtle graphics*. In turtle graphics, you move a robotic pen across a canvas by issuing it simple commands—such as 'move forwards', 'turn left', 'lift up', *etc.* Think about strapping a pen underneath a turtle and walking the turtle around—going forwards, turning left or right, *etc.* Python comes with the `turtle` module that allows you to easily use a turtle to draw. Try this out in the shell:

```
from turtle import *
shape("turtle")    # Shapes the pen like a turtle
forward(50)
left(90)           # The parameter is the number of degrees to turn
forward(50)
left(90)
forward(50)
right(-90)         # This is the same as left(90)
forward(50)
```

As soon as you call `shape`, a *Python Turtle Graphics* window will appear. You can resize this window and keep typing commands into the shell (use `reset()` to re-centre the turtle in the window)—the turtle in the graphics window will move in response to your commands, leaving a trail behind it. You should now have a small square in the graphics window. You can do anything you want with the turtle. Check out the spiral below:

```

from turtle import *
for i in range(1,60):
    width(i/4)
    forward(i)
    right(30)

```

NOTE: If you close the turtle window manually (by clicking on the close button) then it may crash. In this case you will need to restart the shell in Wing. To do this click on the options button (top right of Shell window in Wing) and select *restart shell*.

The commands should be simple enough for you to follow with a pen and paper if you find the code a bit confusing. See `help(RawTurtle)` or the Python Manual² for a complete list of commands you can issue to the turtle.

A simple introductory fractal is a tree (this code is included in `recursion.py`):

```

from turtle import *
from random import *
def tree(size,level):
    if level<>0:
        forward(random()*size)
        x = pos()
        angle=random()*20
        right(angle)
        tree(size*0.8,level-1)
        left(angle)
        setpos(x)
        angle=random()*-20
        right(angle)
        tree(size*0.8,level-1)
        left(angle)
        setpos(x)
tree(100,5)

```

The `fractals.py` module contains the `FractalCanvas` class that sets up a simple user interface for drawing fractals, including a turtle. The `fractal_drawing` module is where we will be implementing sub-classes of the `Fractal` class that actually do some drawing.

> *Complete question 7 in Lab quiz 4.*

The Sierpinski Triangle

The `fractal_drawing` module contains a complete `SierpinskiTriangle` class, and some code at the very bottom for setting up Tkinter. Running the module as-is should produce a window like that shown in Figure 2.

This fractal is known as the *Sierpinski triangle*—built from a repeating set of triangles within other triangles. If you click on the *Draw Fractal* button, it will show you the construction of the triangle. You can also adjust the parameter to the `SierpinskiTriangle` constructor at the bottom of the module to change how many levels of triangles are drawn—but be warned: drawing becomes very slow after around 7 levels; think about how many triangles are being drawn!

The ‘level’, or *degree* of a fractal is a count of how many times it recurses into itself. In this case, it is drawing five levels of triangles stacked inside each-other.

The `draw` method in `SierpinskiTriangle` sets up some basic parameters for the drawing—the co-ordinates of the outer triangle, the colours to use, and resetting the turtle. `draw_triangle` is where the recursive drawing happens:

²<http://docs.python.org/library/turtle.html>

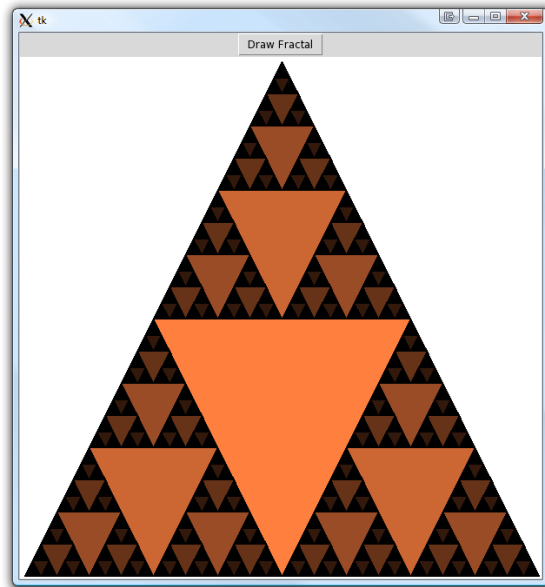


Figure 2: The Sierpinski triangle drawn to 5 levels.

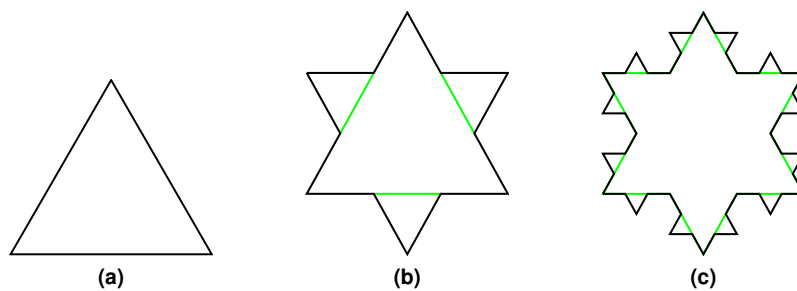


Figure 3: Constructing the Koch snowflake from level 0 to 2.

1. It draws a triangle for the points it has been given (for the first level, these are the points for the outer triangle).
2. If there are more levels to draw...
3. ... calculate the points for the three inner triangles, and call `draw_triangle` to draw them.

Take a moment to experiment with the Sierpinski triangle to understand its drawing code before continuing.

> *Complete question 8 in Lab quiz 4.*

The Koch Snowflake

The *Koch snowflake* is one of the oldest fractal curves; constructed by repeatedly dividing the sides of an equilateral triangle into thirds, and drawing triangles on the middle segment. Figure 3 shows the first few iterations of the snowflake. A snowflake is actually made up of three *Koch curves* drawn at 120° angles; for this exercise we will only draw a single curve (one side of the triangle), but you can implement a snowflake if you wish (see the *Extras* section).

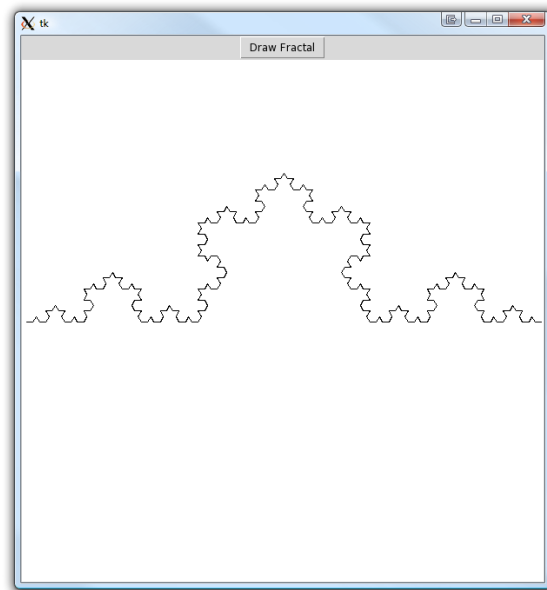


Figure 4: The Koch curve drawn to four levels.

The `fractal_drawing` module contains an incomplete `KochCurve` class that you will complete, based on the procedure described below.

The `draw` method in `KochCurve` is completed for you, and positions the turtle on the screen at the vertical centre of the left edge; it then calls `draw_koch` with the number of levels to draw, and the length of the curve.

The drawing algorithm for the curve is:

1. If we're at the lowest level of the curve (level 0), move forward by the given length (and call `self.update_screen()`).
2. Divide the length by 3.
3. Draw the curve for the next level.
4. Turn left 60° .
5. Draw the curve for the next level.
6. Turn right 120° .
7. Draw the curve for the next level.
8. Turn left 60° .
9. Draw the curve for the next level.

Try tracing these instructions out on a piece of paper for the first couple of levels to get a better feeling for what your turtle will be doing.

Implement these drawing instructions in the `draw_koch` method, using the `self.turtle` object as your turtle. Remember that the levels decrease as you descend into deeper levels of the curve, and you should stop when you hit the *base case*.

Change the code at the bottom of the module to create a new instance of `KochCurve` instead of `SierpinskiTriangle` and run the code. If you implemented everything correctly, you should see something similar to Figure 4 draw on the screen.

> **Complete question 9 in Lab quiz 4.**

Extra Exercises

- Write a recursive function that appends an item to the end of a linked list.
- A *Koch snowflake* is three Koch curves drawn at 120° angles to each other (a triangle). Implement a `KochSnowflake` class that draws this fractal.
- Add some colour to your fractals to indicate some parameter such as the level or age of the line being drawn (for example, `SierpinskiTriangle` draws deeper levels of the triangle in darker shades). You can set the colour of the turtle by calling its `pencolor()` method with the RGB values (0–255), for example: `self.turtle.pencolor(0, 0, 255)` is blue.
- Implement a class for drawing a *Hilbert curve* (see below).
- Implement a class for drawing a *Dragon curve* (see below).

The Hilbert Curve

The *Hilbert curve* is a type of fractal known as a *space-filling curve*—given a square grid of points, it will draw a single line that runs through every point exactly once.

The construction of a Hilbert curve is shown in Figure 5. The first iteration in Figure 5a covers all points in a 2×2 grid. If we double the size of the grid to 4×4 we take the first iteration, and place it in the corners of the 4×4 grid—the lower two are placed as-is, the upper two are rotated 90° left and right, respectively—with connecting lines added between the shapes (Figure 5b). If we want to double the size of the grid again, we repeat this procedure using the last iteration (Figure 5c).

The drawing algorithm for this curve is:

1. Turn to the right.
2. Draw the curve for the next level.
3. Move forward to create a connecting line.
4. Turn to the left.
5. Draw the curve for the next level.
6. Move forward to create a connecting line.
7. Draw the curve for the next level.
8. Turn to the left

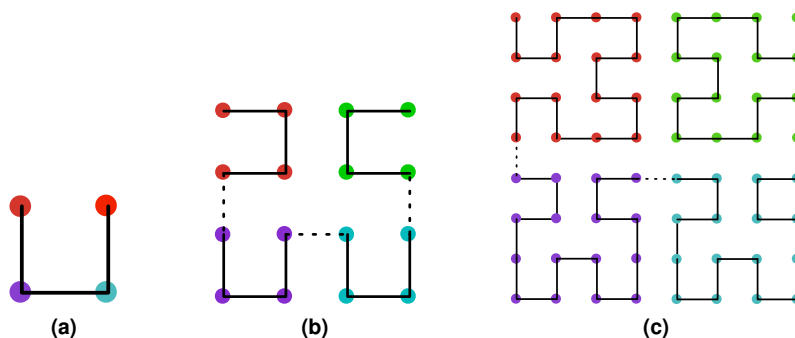


Figure 5: Constructing the Hilbert curve to 3 levels.

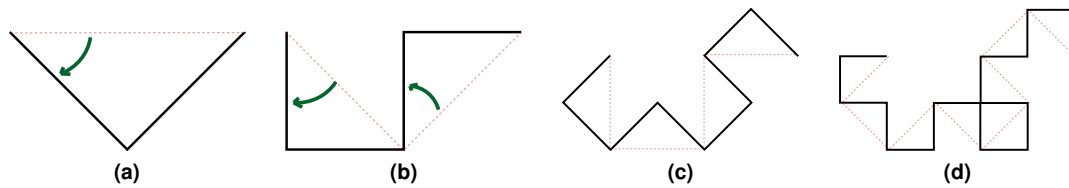


Figure 6: Levels 1–4 of the Dragon curve (level 0 is a straight line).

9. Move forward to create a connecting line.
10. Draw the curve for the next level.
11. Turn to the right.

As you descend into deeper levels, the angle you will need to turn will alternate between 90 and -90 to get the inner curved orientated properly.

The Dragon Curve

The *Dragon curve* is another type of fractal that draws its name from its similarity at high levels to the mythical creature. It is drawn by taking a line and replacing it with two lines at 45° angles to the original line; this process is repeated on the new line segments, alternating the side at which the new lines are drawn on. Figure 6 shows the first few iterations of this process.

The `fractal_drawing` module contains an incomplete `DragonCurve` class that you will complete, based on the procedure described below.

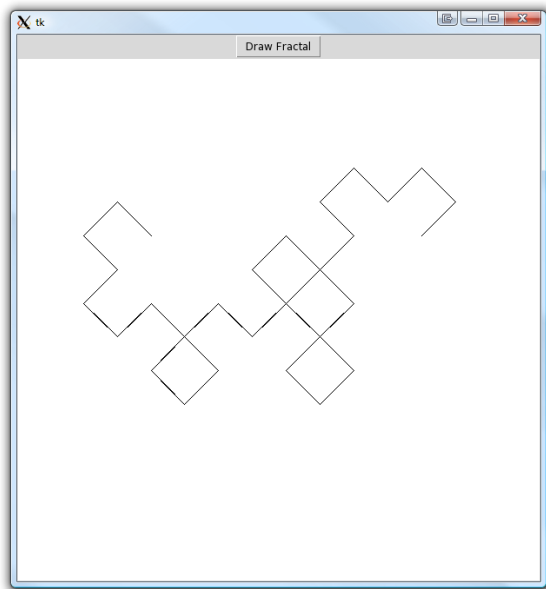
The drawing algorithm for the curve is:

1. If we're at the lowest level of the curve (level 0), move forward by the given length (and call `self.update_screen()`).
2. Divide the length by 1.4 (to fit the curve on the screen).
3. Turn right, using the current value of angle.
4. Draw the next dragon level at 45° .
5. Turn left twice.
6. Draw the next dragon level at -45° .
7. Turn right.

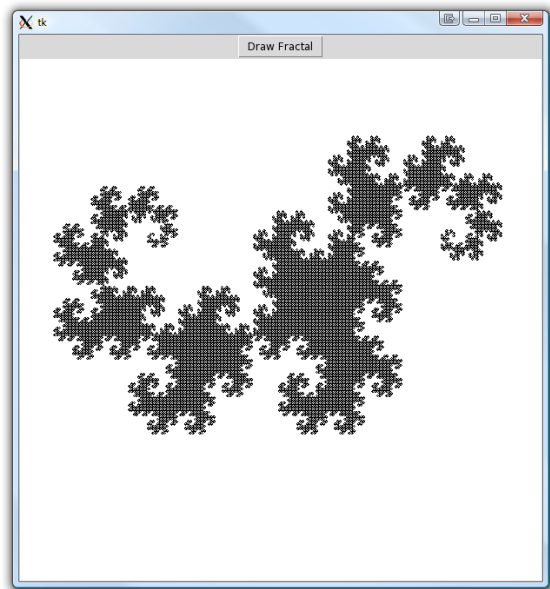
The `draw_dragon` method takes in three parameters: the level of the curve, the length to draw the current segment at, and the angle to turn; this angle determines whether we are dividing current line segment with new lines on either its left or right.

Using the instructions above, implement the `draw_dragon` method using the `self.turtle` object as your turtle. Remember that the levels decrease as you descend deeper into the curve, and to check for the base case.

Change the code at the bottom of the module to create a new instance of `DragonCurve` and run the code. If you implemented everything correctly, you should see something similar to Figure 7a draw on the screen. Experiment with the number of levels; the curve looks best at high levels (around 15), as shown in Figure 7b.



(a)



(b)

Figure 7: The Dragon curve drawn to (a) 5 and (b) 15 levels.