1 Interpreting Strings with Turtle Graphics

We can use strings to instruct the turtle to draw shapes, if we interpret each symbol in the string as a turtle drawing command. Let’s start with these symbols:

- F is forward by a certain distance
- + is left by an angle
- – is right by an angle

Then we can think of a string of symbols as a “shape” for the turtle to draw. If our angle is 90 degrees, then we can draw a square with what string? F+F+F+F.

What information do we need to draw the square? We need the string itself, the angle, and the distance associated with F.

Let’s start a new file (lec22.py) and put all of our string interpretation code in there. We write a function drawString that takes strings of F, +, and - and interprets them as turtle graphics commands. drawString takes the string and two other pieces of information – the distance associated with any F and the angle associated with any + or -. We loop through the string and do what the character tells us.
# lec22.py

```python
import turtle

def drawString(string, distance, angle):
    for char in string:
        if char == 'F':
            turtle.forward(distance)
        elif char == '-':
            turtle.right(angle)
        elif char == '+':
            turtle.left(angle)

if __name__ == '__main__':
    turtle.tracer(False)
    drawString('F+F+F+F+',100, 90)
    turtle.update()
    raw_input('hold on....')
```

## 2 Using L-systems to make Fractals

A fractal is a geometric shape in which each piece of the shape is similar to the whole. The term was coined in 1975 by Mandelbrot. Take a look at the Wikipedia pages on L-systems and fractals for more information and for project ideas.

To make more interesting pictures, such as fractals and plants, we use an L-system that will generate the string for us.

We need the functions from Monday’s lecture in order to get an L-system to generate a string. So let’s open lsystem.py and use that module.

A Koch snowflake can be drawn by the following L-system:

Base $F\rightarrow F--F--F$

Rule $F \rightarrow F+F--F+F$

with an angle of 60 degrees.
We update the lsystem.py code to use the Koch snowflake L-system, and have it generate strings using 0, 1, 2, and more iterations. We copy and paste the output into the testing code in lec22 and observe the results.

The string created after 0 iterations is $F\rightarrow F$, which draws an equilateral triangle (see Figure 1).

![Figure 1: Koch Snowflake with 0 iterations of the production rules](image1.png)

The string created after 1 iteration is $F+F\rightarrow F+F\rightarrow F+F\rightarrow F+F\rightarrow F+F\rightarrow F+F\rightarrow F+F$, which draws the Star of David (see Figure 2).

![Figure 2: Koch Snowflake with 1 iteration of the production rules](image2.png)
After two iterations, it has ballooned to $F+F--F+F+F--F+F--F+F--F+F+F+$
$F--F+F--F+F--F+F+F--F+F--F+F--F+F+F--F+F+F--F+F+F--F+F+F+
F+F--F+F--F+F--F+F+F--F+F--F+F+F--F+F--F+F+F--F+F+F$.

Figure 3: Koch Snowflake with 2 iterations of the production rules

Notice that if you increase the number of iterations, you should decrease the distance.

To draw a snowflake with 5 iterations, I reduce the distance to 1, and it still takes up much of my turtle window. The snowflake is shown in Figure 4. It
is difficult to see the change in size in these notes, but it will become readily apparent when you run the code yourself.

Figure 4: Koch Snowflake with 5 iterations of the production rules

3 Adding Branching

I want to draw a plant, with branches. Then I draw up one branch. I want to draw the next. I need to get the turtle back to the base of the branch. To do that I need to implement a “memory”.

We add two new characters to the alphabet:
[ is save the turtle state (i.e. position and orientation)

] is restore the turtle state (i.e. position and orientation). In particular, restore the last state that was saved.

With the string “F[+F]F”, we draw forward, save our state, turn left 25 degrees, draw that branch by going forward, then return to the base of that branch (and our original heading) and draw forward again (see Figure 5).

![Figure 5: “Plant” with a single branch](image)

We need to add support for the interpretation of the state-saving. We do this by saving the position and heading in a “stack”. A stack is a list, but with special rules – the last item added is always the first item removed.

Let’s look at a longer plant string: “F[+F[+F]F]F” By storing the positions in a stack, we return to the most recent branch point every time we encounter a “]”. The picture is shown in Figure 6.

![Figure 6: “Plant” with two branches](image)

To implement the turtle save/restore commands, we use a Python list as our stack. We insert an object onto the stack using list.append() and remove an object from the stack using list.pop()
The updated code for drawString is shown here:

```python
def drawString(distance, angle, string):
    stack = []
    for char in string:
        if char == 'F':
            turtle.forward(distance)
        elif char == '-':
            turtle.right(angle)
        elif char == '+':
            turtle.left(angle)
        elif char == '[':
            stack.append(turtle.position())
            stack.append(turtle.heading())
        elif char == ']':
            turtle.up()
            turtle.setheading(stack.pop())
            turtle.goto(stack.pop())
            turtle.down()
```

We test the new code with the above examples (using an angle of 25 degrees) (again, see Figures 5 and 6).