Turtles and Strings and L-Systems

This section describes a much more interested example of string iteration and the accumulator pattern. Even though it seems like we are doing something that is much more complex, the basic processing is the same as was shown in the previous sections.

In 1968 Astrid Lindenmayer, a biologist, invented a formal system that provides a mathematical description of plant growth known as an **L-system**. L-systems were designed to model the growth of biological systems. You can think of L-systems as containing the instructions for how a single cell can grow into a complex organism. L-systems can be used to specify the **rules** for all kinds of interesting patterns. In our case, we are going to use them to specify the rules for drawing pictures.

The rules of an L-system are really a set of instructions for transforming one string into a new string. After a number of these string transformations are complete, the string contains a set of instructions. Our plan is to let these instructions direct a turtle as it draws a picture.

To begin, we will look at an example set of rules:

A	Axiom
A -> B	Rule 1 Change A to B
B -> AB	Rule 2 Change B to AB

Each rule set contains an axiom which represents the starting point in the transformations that will follow. The rules are of the form:

```
left hand side -> right hand side
```

where the left and side is a single symbol and the right had side is a sequence of symbols. You can think of both sides as being simple strings. The way the rules are used is to replace occurrences of the left hand side with the corresponding right hand side.

Now lets look at these simple rules in action, starting with the string A:

```
A
B      Apply Rule 1  (A is replaced by B)
AB     Apply Rule 2  (B is replaced by AB)
BAB    Apply Rule 1 to A then Rule 2 to B
ABBAB  Apply Rule 2 to B, Rule 1 to A, and Rule 2 to B
```

Notice that each line represents a new transformation for entire string. Each character that matches a left-hand side of a rule in the original has been replaced by the corresponding right-hand side of that same rule. After doing the replacement for each character in the original, we have one transformation.

So how would we encode these rules in a Python program? There are a couple of very important things to note here:

1. Rules are very much like if statements.
2. We are going to start with a string and iterate over each of its characters.
3. As we apply the rules to one string we leave that string alone and create a brand new string using the accumulator pattern. When we are all done with the original we replace it with the new string.

Lets look at a simple Python program that implements the example set of rules described above.

```

    return newstr


def createLSystem(numIters,axiom):
    startString = axiom
    endString = ""
    for i in range(numIters):
        endString = processString(startString)
        startString = endString

    return endString

print(createLSystem(4,"A"))

```

ActiveCode: 1 (string_lsys1)

Run

```
ABBAB
```

Try running the example above with different values for the `numIters` parameter. You should see that for values 1, 2, 3, and 4, the strings generated follow the example above exactly.

One of the nice things about the program above is that if you want to implement a different set of rules, you don’t need to re-write the entire program. All you need to do is re-write the applyRules function.

Suppose you had the following rules:

A	Axiom
A -> BAB	Rule 1 Change A to BAB

What kind of a string would these rules create? Modify the program above to implement the rule.

Now lets look at a real L-system that implements a famous drawing. This L-system has just two rules:

F	Axiom
F -> F-F++F-F	Rule 1

This L-system uses symbols that will have special meaning when we use them later with the turtle to draw a picture.

F	Go forward by some number of units
B	Go backward by some number of units
-	Turn left by some degrees
+	Turn right by some degrees

Here is the `applyRules` function for this L-system.

```
def applyRules(ch):
    newstr = ""
    if ch == 'F':
        newstr = 'F-F++F-F'   # Rule 1
    else:
        newstr = ch    # no rules apply so keep the character

    return newstr
```

Pretty simple so far. As you can imagine this string will get pretty long with a few applications of the rules. You might try to expand the string a couple of times on your own just to see.

The last step is to take the final string and turn it into a picture. Lets assume that we are always going to go forward or backward by 5 units. In addition we will also assume that when the turtle turns left or right we’ll turn by 60 degrees. Now look at the string `F-F++F-F` . You might try to us the explanation above to show the resulting picture that this simple string represents. At this point its not a very exciting drawing, but once we expand it a few times it will get a lot more interesting.

To create a Python function to draw a string we will write a function called `drawLsystem` The function will take four parameters:

- A turtle to do the drawing
- An expanded string that contains the results of expanding the rules above.
- An angle to turn
- A distance to move forward or backward

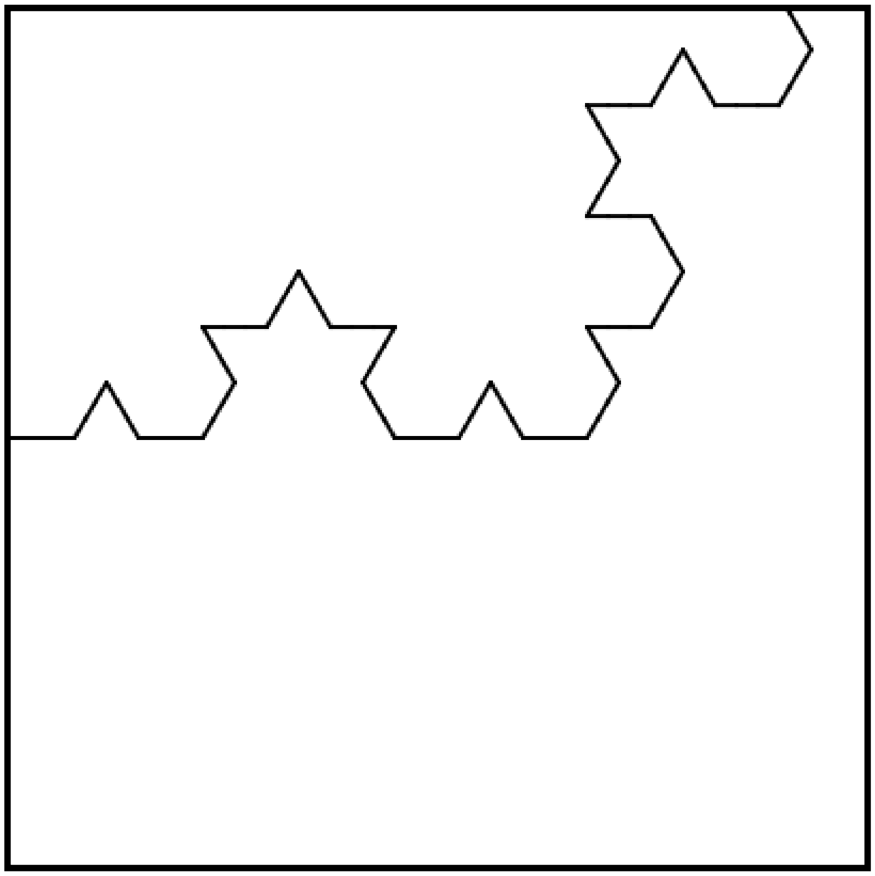
```
def drawLsystem(aTurtle,instructions,angle,distance):
    for cmd in instructions:
        if cmd == 'F':
            aTurtle.forward(distance)
        elif cmd == 'B':
            aTurtle.backward(distance)
        elif cmd == '+':
            aTurtle.right(angle)
        elif cmd == '-':
            aTurtle.left(angle)
        else:
            print('Error:', cmd, 'is an unknown command')
```

Here is the complete program in activecode. The `main` function first creates the L-system string and then it creates a turtle and passes it and the string to the drawing function.

```
    inst = createLSystem(4,"F")   #create the string
    print(inst)
    t = turtle.Turtle()           #create the turtle
    wn = turtle.Screen()

    t.up()
    t.back(200)
    t.down()
    t.speed(0)
    drawLsystem(t,inst,60,30)      #draw the picture
                                  #angle 60, segment length 5
    wn.exitonclick()

main()

```

ActiveCode: 2 (strings_lys2)

Run



```
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++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-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++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++F-F++F-F-F-F++F-F-F-F+
+F-F-F-F++F-F++F-F++F-F-F-F++F-F
```

Feel free to try some different angles and segment lengths to see how the drawing changes.

Looping and counting

We will finish this chapter with a few more examples that show variations on the theme of iteration through the characters of a string. We will implement a few of the methods that we described earlier to show how they can be done.

The following program counts the number of times a particular letter, `aChar`, appears in a string. It is another example of the accumulator pattern that we have seen in previous chapters.

```python
def count(text, aChar):
    lettercount = 0
    for c in text:
        if c == aChar:
            lettercount = lettercount + 1
    return lettercount

print(count("banana","a"))

```

ActiveCode: 3 (chp08_fun2)

Run

```
3
```

The function `count` takes a string as its parameter. The `for` statement iterates through each character in the string and checks to see if the character is equal to the value of `aChar`. If so, the counting variable, `lettercount`, is incremented by one. When all characters have been processed, the `lettercount` is returned.

A `find` function

Here is an implementation for the `find` method.

```python
def find(astring, achar):
    """
      Find and return the index of achar in astring.
      Return -1 if achar does not occur in astring.
    """
    ix = 0
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
```

ActiveCode: 4 (ch08_run3)

Run

```
3
0
6
-1
```

In a sense, `find` is the opposite of the indexing operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears for the first time. If the character is not found, the function returns `-1`.

The `while` loop in this example uses a slightly more complex condition than we have seen in previous programs. Here there are two parts to the condition. We want to keep going if there are more characters to look through and we want to keep going if we have not found what we are looking for. The variable `found` is a boolean variable that keeps track of whether we have found the character we are searching for. It is initialized to *False*. If we find the character, we reassign `found` to *True*.

The other part of the condition is the same as we used previously to traverse the characters of the string. Since we have now combined these two parts with a logical `and`, it is necessary for them both to be *True* to continue iterating. If one part fails, the condition fails and the iteration stops.

When the iteration stops, we simply ask a question to find out why and then return the proper value.

Note

This pattern of computation is sometimes called a eureka traversal because as soon as we find what we are looking for, we can cry Eureka! and stop looking. The way we stop looking is by setting `found` to True which causes the condition to fail.

Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the `find` function, adding a third parameter for the starting position in the search string:

```
    """
    ix = start
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

print(find2('banana', 'a', 2))

```

ActiveCode: 5 (ch08_fun4)

Run

```
3
```

The call `find2('banana', 'a', 2)` now returns `3`, the index of the first occurrence of ‘a’ in ‘banana’ after index 2. What does `find2('banana', 'n', 3)` return? If you said, 4, there is a good chance you understand how `find2` works. Try it.

Better still, we can combine `find` and `find2` using an **optional parameter**.

```
    """
    ix = start
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

print(find3('banana', 'a', 2))

```

ActiveCode: 6 (chp08_fun5)

Run

```
3
```

The call `find3('banana', 'a', 2)` to this version of `find` behaves just like `find2` , while in the call `find3('banana', 'a')` , `start` will be set to the **default value** of `0` .

Adding another optional parameter to `find` makes it search from a starting position, up to but not including the end position.

```
            else:
                ix = ix + 1
        if found:
            return ix
        else:
            return -1

ss = "Python strings have some interesting methods."

print(find4(ss, 's'))
print(find4(ss, 's', 7))
print(find4(ss, 's', 8))
print(find4(ss, 's', 8, 13))
print(find4(ss, '.'))

```

ActiveCode: 7 (chp08_fun6)

Run

```
7
7
13
-1
44
```

The optional value for `end` is interesting. We give it a default value `None` if the caller does not supply any argument. In the body of the function we test what `end` is and if the caller did not supply any argument, we reassign `end` to be the length of the string. If the caller has supplied an argument for `end`, however, the caller's value will be used in the loop.

The semantics of `start` and `end` in this function are precisely the same as they are in the `range` function.

Character classification

It is often helpful to examine a character and test whether it is upper- or lowercase, or whether it is a character or a digit. The `string` module provides several constants that are useful for these purposes. One of these, `string.digits` is equivalent to "0123456789". It can be used to check if a character is a digit using the `in` operator.

The string `string.ascii_lowercase` contains all of the ascii letters that the system considers to be lowercase. Similarly, `string.ascii_uppercase` contains all of the uppercase letters. `string.punctuation` comprises all the characters considered to be punctuation. Try the following and see what you get.

```
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.punctuation)
```

For more information consult the `string` module documentaiton (see Global Module Index (http://docs.python.org/py3k/py-modindex.html)).

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.punctuation)
```

ActiveCode: 8 (scratch_08_04)

Run

Error

```
NameError: name 'string' is not defined on line 1
```

Description

A name error almost always means that you have used a variable before it has a value. Often this may be a simple typo, so check the spelling carefully.

To Fix

Check the right hand side of assignment statements and your function calls, this is the most likely place for a NameError to be found.

Summary

This chapter introduced a lot of new ideas. The following summary may prove helpful in remembering what you learned.

indexing (`[]`)

Access a single character in a string using its position (starting from 0). Example: `'This'[2]` evaluates to `'i'` .

length function (`len`)

Returns the number of characters in a string. Example: `len('happy')` evaluates to `5` .

for loop traversal (`for`)

Traversing a string means accessing each character in the string, one at a time. For example, the following for loop:

```
for ix in 'Example':
    ...
```

executes the body of the loop 7 times with different values of ix each time.

slicing (`[:]`)

A *slice* is a substring of a string. Example: `'bananas and cream'[3:6]` evaluates to `ana` (so does `'bananas and cream'[1:4]`).

string comparison (`>, <, >=, <=, ==, !=`)

The six common comparision operators work with strings, evaluating according to lexigraphical order (http://en.wikipedia.org/wiki/Lexicographic_order). Examples: `'apple' < 'banana'` evaluates to `True` . `'Zeta' < 'Appricot'` evaluates to `False` . `'Zebra' <= 'aardvark'` evaluates to `True` because all upper case letters precede lower case letters.

in and not in operator (`in` , `not in`)

The `in` operator tests whether one string is contained inside another string. Examples: `'heck' in "I'll be checking for you."` evaluates to `True` . `'cheese' in "I'll be checking for you."` evaluates to `False` .