# The Return of L-Systems

Lets return the the L-systems we introduced in the previous chapter and introduce a very interesting new feature that requires the use of lists.

Suppose we have the following grammar:

```
X
X --> F[-X]+X
F --> FF
```

This L-system looks very similar to the old L-system except that we've added one change. We've added the characters '[' and ']'. The meaning of these characters adds a very interesting new dimension to our L-Systems. The '[' character indicates that we want to save the state of our turtle, namely its position and its heading so that we can come back to this position later. The ']' tells the turtle to warp to the most recently saved position. The way that we will accomplish this is to use lists. We can save the heading and position of the turtle as a list of 3 elements. `[heading x y]` The first index position in the list holds the heading, the second index position in the list holds the x coordinate, and the third index position holds the y coordinate.

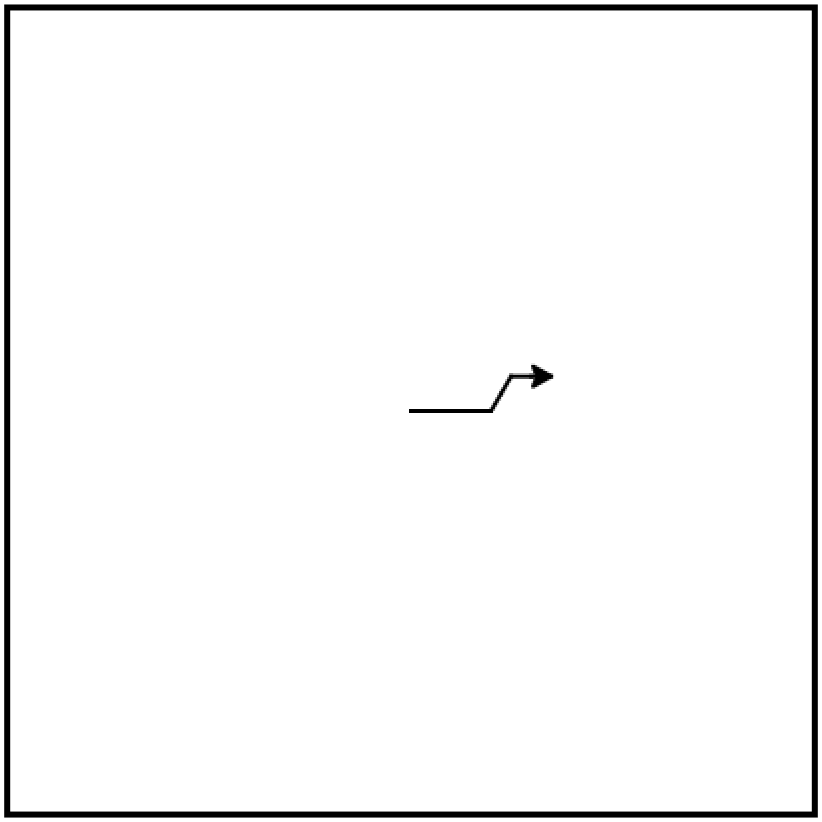
Now, if we create an empty list and every time we see a '[' we append the list that contains `[heading, x, y]` we create a history of saved places the turtle has been where the most recently saved location will always be at the end of the list. When we find a ']' in the string we use the pop function to remove the the most recently appended information.

Lets modify our `drawLsystem` function to begin to implement this new behavior.

```
        elif cmd == '[':
            savedInfoList.append([aTurtle.heading(),aTurtle.xcor(),aTurtle.ycor()])
            print(savedInfoList)
        elif cmd == ']':
            newInfo = savedInfoList.pop()
            print(newInfo)
            print(savedInfoList)
        else:
            print('Error:', cmd, 'is an unknown command')

t = turtle.Turtle()
inst = "FF[-F[-X]+X]+F[-X]+X"
drawLsystem(t,inst,60,20)

```

**ActiveCode: 1** (list_lsys1)

Run



```
[[0, 40, 0]]
[[0, 40, 0], [59.99999999999999, 50, 17.32050807568877]]
Error: X is an unknown command
[59.99999999999999, 50, 17.32050807568877]
[[0, 40, 0]]
Error: X is an unknown command
[0, 40, 0]
[]
[[0, 70, 17.32050807568877]]
Error: X is an unknown command
[0, 70, 17.32050807568877]
[]
Error: X is an unknown command
```

When we run this example we can see that the picture is not very interesting, but notice what gets printed out, and how the saved information about the turtle gets added and removed from the end of the list. In the next example we'll make use of the information from the list to save and restore the turtle's position and heading when needed. We'll use a longer example here so you get an idea of what the kind of drawing the L-System can really make.

```
16 savedInfoList.append([aTurtle.heading(),aTurtle.xcor(),aTurtle.ycor()])
17             print(savedInfoList)
18         elif cmd == ']':
19             newInfo = savedInfoList.pop()
20             aTurtle.setheading(newInfo[0])
21             aTurtle.setposition(newInfo[1],newInfo[2])
22         else:
23             print('Error:', cmd, 'is an unknown command')
24
25 t = turtle.Turtle()
26 inst = "FF[-F[-X]+X]+F[-X]+X"
27 t.setposition(0,-200)
28 t.left(90)
29 drawLsystem(t,inst,30,2)
30
```

**ActiveCode: 2** (list_lsys2)

Run

## Error

```
ParseError: bad input on line 16
```

## Description

A parse error means that Python does not understand the syntax on the line the error message points out. Common examples are forgetting commas beteween arguments or forgetting a : on a for statement

## To Fix

To fix a parse error you just need to look carefully at the line with the error and possibly the line before it. Make sure it conforms to all of Python's rules.

Rather than use the `inst` string supplied here, use the code from the string chapter, and write your own applyRules function to implement this L-system. This example only uses 6 expansions. Try it out with a larger number of expansions. You may also want to try this example with different values for the angle and distance parameters.

## Append versus Concatenate

The `append` method adds a new item to the end of a list. It is also possible to add a new item to the end of a list by using the concatenation operator. However, you need to be careful.

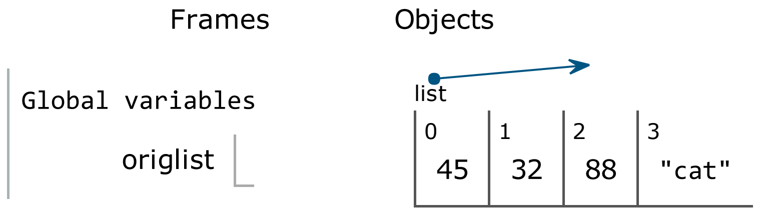
Consider the following example. The original list has 3 integers. We want to add the word “cat” to the end of the list.

```
origlist = [45,32,88]

origlist.append("cat")
```

<< First   < Back   Program terminated   Forward >   Last >>

➡ line that has just executed

➡ next line to execute



**CodeLens: 1 (appcon1)**

Here we have used `append` which simply modifies the list. In order to use concatenation, we need to write an assignment statement that uses the accumulator pattern:

```
origlist = origlist + ["cat"]
```

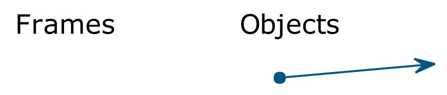
Note that the word “cat” needs to be placed in a list since the concatenation operator needs two lists to do its work.

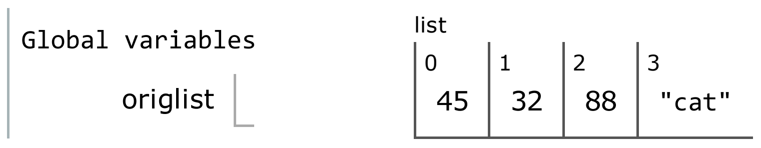
```
origlist = [45,32,88]

origlist = origlist + ["cat"]
```

<< First   < Back   Program terminated   Forward >   Last >>

➡ line that has just executed

➡ next line to execute





**CodeLens: 2 (appcon2)**

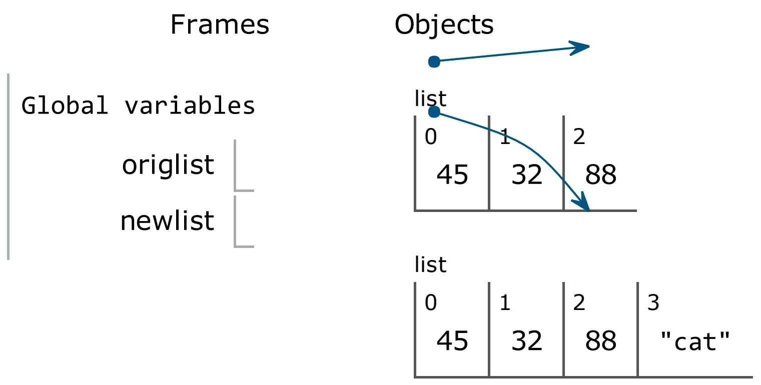
It is also important to realize that with append, the original list is simply modified. On the other hand, with concatenation, an entirely new list is created. This can be seen in the following codelens example where `newlist` refers to a list which is a copy of the original list, `origlist`, with the new item “cat” added to the end. `origlist` still contains the three values it did before the concatenation. This is why the assignment operation is necessary as part of the accumulator pattern.

```
origlist = [45,32,88]

newlist = origlist + ["cat"]
```

<< First  < Back  Program terminated  Forward >  Last >>

➡ line that has just executed

➡ next line to execute



**CodeLens: 3 (appcon3)**

**Check you understanding**

## Lists and `for` loops

It is also possible to perform **list traversal** using iteration by item as well as iteration by index.

```python
fruits = ["apple","orange","banana","cherry"]

for afruit in fruits:     # by item
    print(afruit)

```

**ActiveCode: 3** (chp09_03a)

Run

```
apple
orange
banana
cherry
```

It almost reads like natural language: For (every) fruit in (the list of) fruits, print (the name of the) fruit.

We can also use the indices to access the items in an iterative fashion.

```python
fruits = ["apple","orange","banana","cherry"]

for position in range(len(fruits)):     # by index
    print(fruits[position])

```

**ActiveCode: 4** (chp09_03b)

Run

```
apple
orange
banana
cherry
```

In this example, each time through the loop, the variable `position` is used as an index into the list, printing the `position`-eth element. Note that we used `len` as the upper bound on the range so that we can iterate correctly no matter how many items are in the list.

Any sequence expression can be used in a `for` loop. For example, the `range` function returns a sequence of integers.

```python
for number in range(20):
    if number % 3 == 0:
        print(number)

```

**ActiveCode: 5** (chp09_for3)

Run

```
0
3
6
9
12
15
18
```

This example prints all the multiples of 3 between 0 and 19.

Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements as you go. The following code squares all the numbers from `1` to `5` using iteration by position.

```python
numbers = [1, 2, 3, 4, 5]
print(numbers)

for i in range(len(numbers)):
    numbers[i] = numbers[i]**2

print(numbers)

```

**ActiveCode: 6** (chp09_for4)

Run

```
[1, 2, 3, 4, 5]
[1, 4, 9, 16, 25]
```

Take a moment to think about `range(len(numbers))` until you understand how it works. We are interested here in both the *value* and its *index* within the list, so that we can assign a new value to it.

**Check your understanding**

## Using Lists as Parameters

Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**. Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing. For example, the function below takes a list as an argument and multiplies each element in the list by 2:

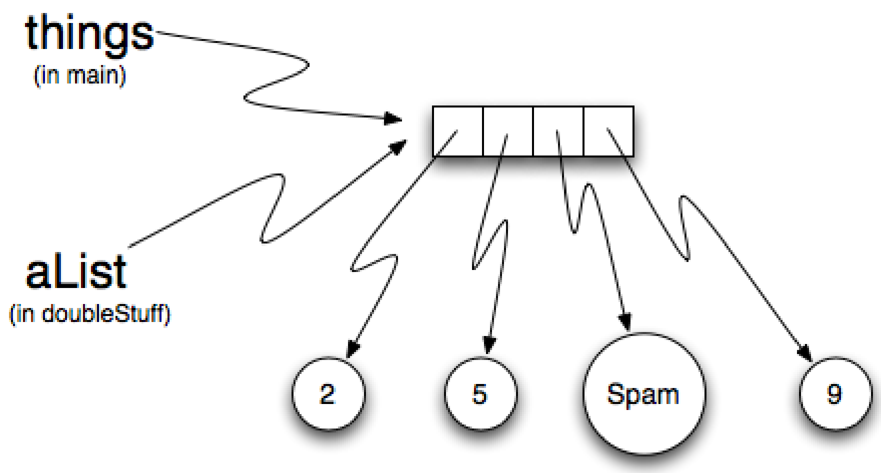
```
def doubleStuff(aList):
    """ Overwrite each element in aList with double its value. """
    for position in range(len(aList)):
        aList[position] = 2 * aList[position]

things = [2, 5, 9]
print(things)
doubleStuff(things)
print(things)

```

**ActiveCode: 7** (chp09_parm1)

Run

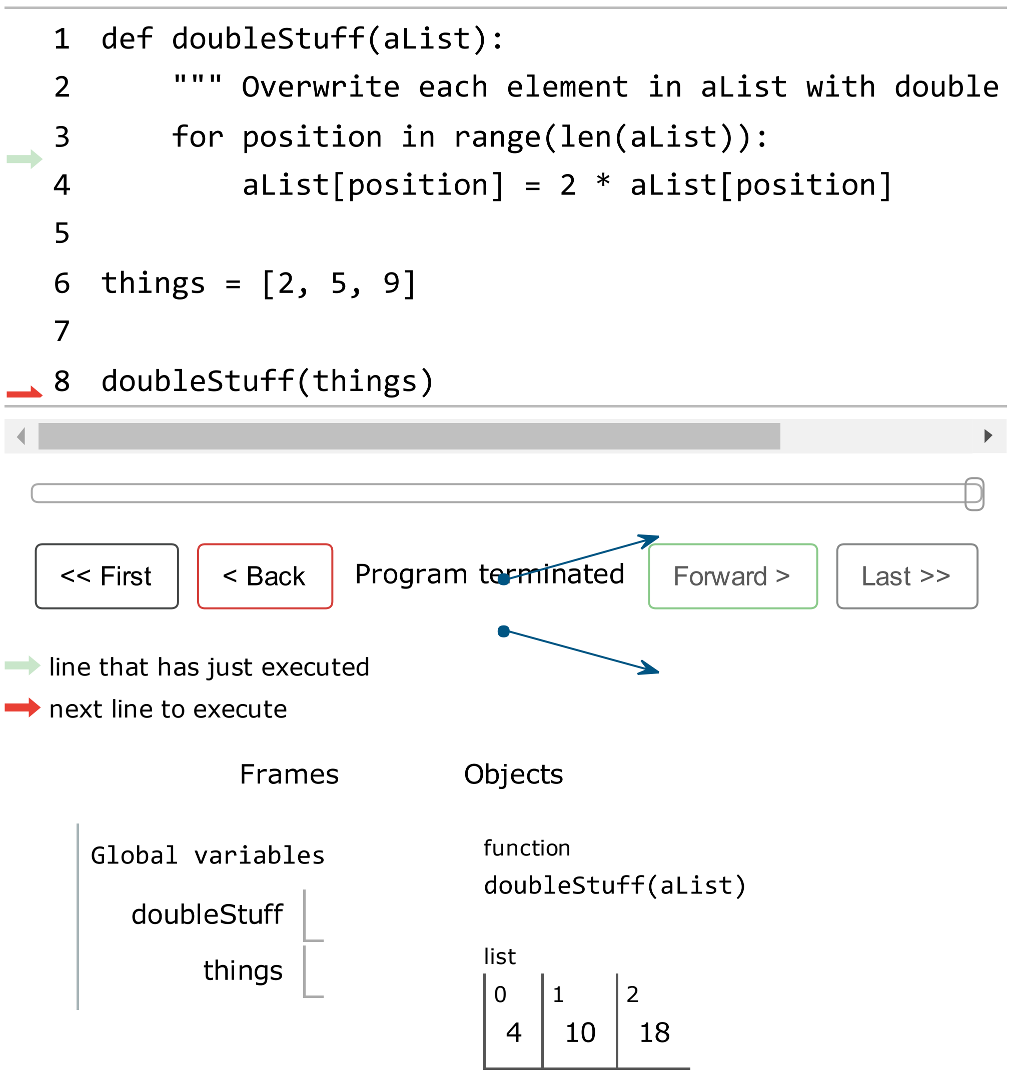
```
[2, 5, 9]
[4, 10, 18]
```

The parameter `aList` and the variable `things` are aliases for the same object.



Since the list object is shared by two references, there is only one copy. If a function modifies the elements of a list parameter, the caller sees the change since the change is occurring to the original.

This can be easily seen in codelens. Note that after the call to `doubleStuff`, the formal parameter `aList` refers to the same object as the actual parameter `things`. There is only one copy of the list object itself.



```
def doubleStuff(aList):
    """ Overwrite each element in aList with double
    for position in range(len(aList)):
        aList[position] = 2 * aList[position]

things = [2, 5, 9]

doubleStuff(things)
```

**CodeLens: 4 (chp09_parm1_trace)**

## Pure Functions

A **pure function** does not produce side effects. It communicates with the calling program only through parameters (which it does not modify) and a return value. Here is the `doubleStuff` function from the previous section written as a pure function. To use the pure function version of `double_stuff` to modify `things`, you would assign the return value back to `things`.

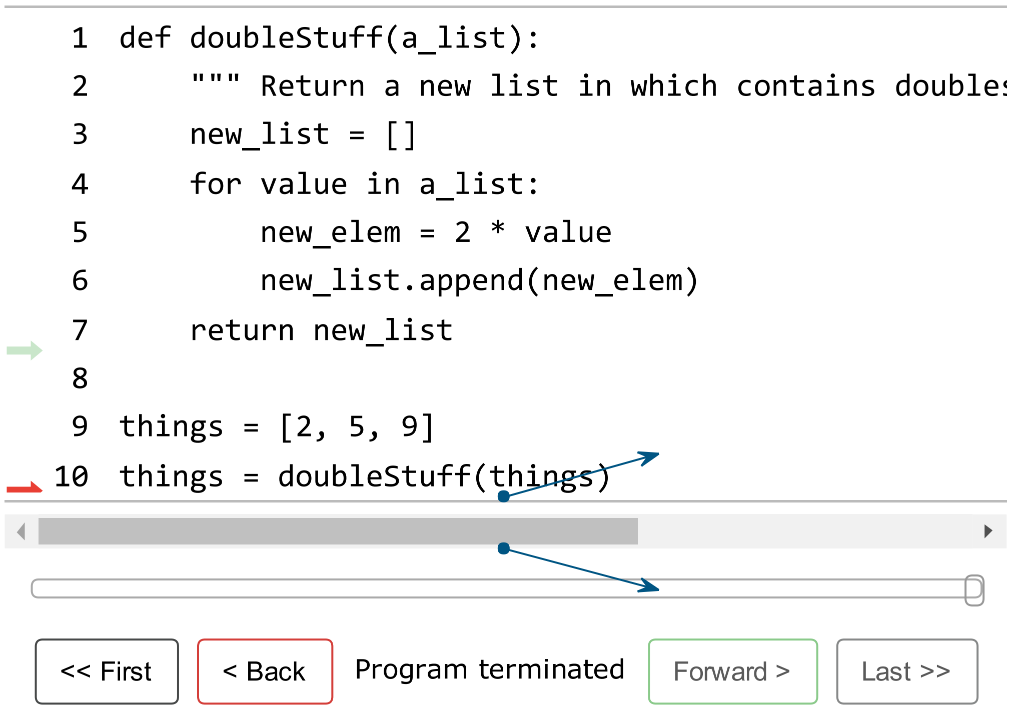
```python
def doubleStuff(a_list):
    """ Return a new list in which contains doubles of the elements in a_list. """
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
print(things)
things = doubleStuff(things)
print(things)
```

**ActiveCode: 8** (ch09_mod2)

Run

```
[2, 5, 9]
[4, 10, 18]
```

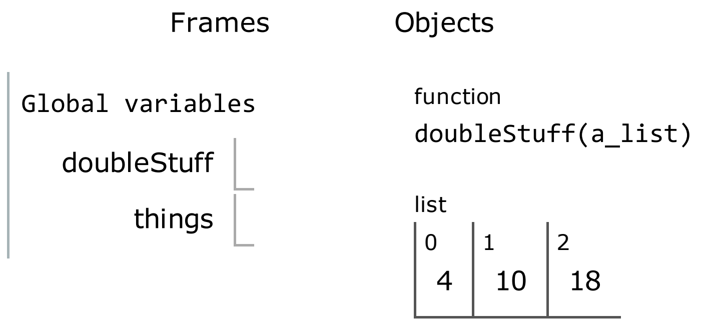
Once again, codelens helps us to see the actual references and objects as they are passed and returned.



```python
def doubleStuff(a_list):
    """ Return a new list in which contains double
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
things = doubleStuff(things)
```

line that has just executed

next line to execute



**CodeLens: 5 (ch09_mod3)**

## Which is Better?

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a *functional programming style*.

## Functions that Produce Lists

The pure version of `doubleStuff` above made use of an important **pattern** for your toolbox. Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
   create a new element
   append it to result
return the result
```

Let us show another use of this pattern. Assume you already have a function `is_prime(x)` that can test if x is prime. Now, write a function to return a list of all prime numbers less than n:

```
def primes_upto(n):
    """ Return a list of all prime numbers less than n. """
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
    return result
```

## List Comprehensions

The previous example creates a list from a sequence of values based on some selection criteria. An easy way to do this type of processing in Python is to use a **list comprehension**. List comprehensions are concise ways to create lists. The general syntax is:

```
[<expression> for <item> in <sequence> if  <condition>]
```

where the if clause is optional. For example,

```
mylist = [1,2,3,4,5]

yourlist = [item ** 2 for item in mylist]

print(yourlist)

```

**ActiveCode: 9** (listcomp1)

Run

```
[1, 4, 9, 16, 25]
```

The expression describes each element of the list that is being built. The `for` clause iterates thru each item in a sequence. The items are filtered by the `if` clause if there is one. In the example above, the `for` statement lets `item` take on all the values in the list `mylist`. Each item is then squared before it is added to the list that is being built. The result is a list of squares of the values in `mylist`.

To write the `primes_upto` function we will use the `is_prime` function to filter the sequence of integers coming from the `range` function. In other words, for every integer from 2 up to but not including `n`, if the integer is prime, keep it in the list.

```
def primes_upto(n):
    """ Return a list of all prime numbers less than n using a list comprehension. """

    result = [num for num in range(2,n) if is_prime(num)]
    return result
```

**Note**

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

```


```

**ActiveCode: 10** (scratch_09_06)

Run

**Check your understanding**

## Nested Lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list. If we print( `nested[3]` ), we get `[10, 20]` . To extract an element from the nested list, we can proceed in two steps. First, extract the nested list, then extract the item of interest. It is also possible to combine those steps using bracket operators that evaluate from left to right.

```
nested = ["hello", 2.0, 5, [10, 20]]
innerlist = nested[3]
print(innerlist)
item = innerlist[1]
print(item)

print(nested[3][1])

```

**ActiveCode: 11** (chp09_nest)

Run

```
[10, 20]
20
20
```

**Check your understanding**