9.8 PROJECTS

*Project 9.1 Lindenmayer's beautiful plants

For this project, we assume you have read Section 9.6.

Aristid Lindenmayer was specifically interested in modeling the branching behavior of plants. To accomplish this, we need to introduce two more symbols: `[` and `]`. For example, consider the following L-system:

Axiom:	`X`
Productions:	`X` → `F[-X]+X`
	`F` → `FF`
Angle:	30 degrees

These two new symbols involve the use of a simple data structure called a *stack*. A stack is simply a list in which we only append to and delete from one end. The append operation is called a *push* and the delete operation is called a *pop* (hence the name of the list `pop` method). For example, consider the following sequence of operations on a hypothetical stack object named `stack`, visualized in Figure 1.

1. `stack.push(1)`
2. `stack.push(2)`
3. `x = stack.pop()`
4. `stack.push(3)`
5. `y = stack.pop()`
6. `z = stack.pop()`

Although we implement a stack as a restricted list, it is usually visualized as a vertical stack of items in which we always push and pop items from the top. The result of the first push operation above results in the leftmost picture in Figure 1. After the second push operation, we have two numbers on the stack, with the second number on top of the first, as in the second picture in Figure 1. The third operation, a pop, removes the top item, which is then assigned to the variable name `x`. The fourth operation pushes the value 3 on the top of the stack. The fifth operation pops this value and assigns it to the variable name `y`. Finally, the bottom value is popped

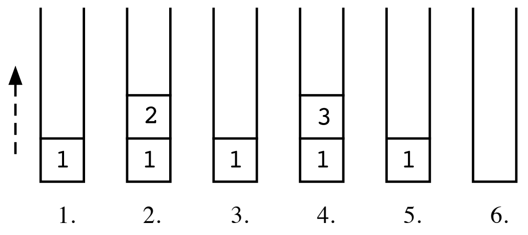


Figure 1 The results of a sequence of stack operations.

and assigned to the variable name `z`. The final values of `x`, `y`, and `z` are `2`, `3`, and `1`, respectively.

In Python, we can represent the stack as an initially empty list, implement the push operation as an `append` and implement the pop operation as a `pop` with no arguments (which defaults to deleting the last item in the list). So the equivalent sequence of Python statements is:

```
stack = [ ]        # empty stack; stack is now [ ]
stack.append(1)    # push 1;      stack is now [1]
stack.append(2)    # push 2;      stack is now [1, 2]
x = stack.pop()    # x is now 2;  stack is now [1]
stack.append(3)    # push 3;      stack is now [1, 3]
y = stack.pop()    # y is now 3;  stack is now [1]
z = stack.pop()    # z is now 1;  stack is now [ ]
```

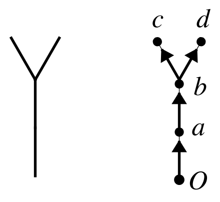
In a Lindenmayer system, the `[` symbol represents a push operation and the `]` symbol represents a pop operation. More specifically,

- `[` means "push the turtle's current position and heading on a stack," and
- `]` means "pop a position and heading from the stack and set the turtle's current position and heading to these values."

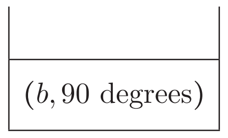
Let's now return to the Lindenmayer system above. Applying the productions of this Lindenmayer system twice results in the following string.

$$\texttt{X} \Rightarrow \texttt{F[-X]+X} \Rightarrow \texttt{FF[-F[-X]+X]+F[-X]+X}$$

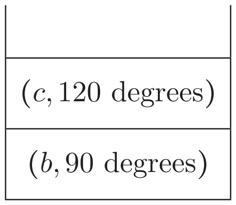
The `X` symbols are used only in the derivation process and do not have any meaning for turtle graphics, so we simply skip them when we are drawing. So the string `FF[-F[-X]+X]+F[-X]+X` represents the simple "tree" below. On the left is a drawing of the tree; on the right is a schematic we will use to explain how it was drawn.



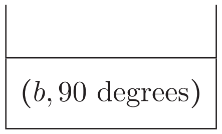
The turtle starts at the origin, marked O, with a heading of 90 degrees (north). The first two `F` symbols move the turtle forward from the origin to point a and then point b. The next symbol, `[`, means that we push the current position and heading $(b, 90$ degrees$)$ on the stack.



The next two symbols, `-F`, turn the turtle left 30 degrees (to a heading of 120 degrees) and move it forward, to point c. The next symbol is another `[`, which pushes the current position and heading, $(c, 120$ degrees$)$, on the stack. So now the stack contains two items—$(b, 90$ degrees$)$ and $(c, 120$ degrees$)$—with the last item on top.



The next three symbols, `-X]`, turn the turtle left another 30 degrees (to a heading of 150 degrees), but then restore its heading to 120 degrees by popping $(c, 120$ degrees$)$ from the stack.



The next three symbols, `+X]`, turn the turtle 30 degrees to the right (to a heading of 90 degrees), but then pop $(b, 90$ degrees$)$ from the stack, moving the turtle back to point b, heading north.

(So, in effect, the previous six symbols, `[-X]+X` did nothing.) The next two symbols, `+F`, turn the turtle 30 degrees to the right (to a heading of 60 degrees) and move it forward to point d. Similar to before, the last six symbols, `[-X]+X`, while pushing states onto the stack, have no visible effect.

Continued applications of the productions in the L-system above will produce strings that draw the same sequence of trees that we created in Section 9.1. More involved L-systems will produce much more interesting trees. For example, the following two L-systems produce the trees in Figure 2.

Axiom:	`X`	Axiom:	`F`
Productions:	`X → F-[[X]+X]+F[+FX]-X`	Production:	`F → FF-[-F+F+F]+[+F-F-F]`
	`F → FF`	Angle:	22.5 degrees
Angle:	25 degrees		

Question 9.1.1 *Using an angle of* 20 *degrees, draw the figure corresponding to the string*

```
FF-[-F+F+F]+[+F-F-F]
```

(Graph paper might make this easier.)

Question 9.1.2 *Using an angle of* 30 *degrees, draw the figure corresponding to the string*

```
FF-[[FF+F]+FF+F]+FF[+FFFF+F]-FF+F
```

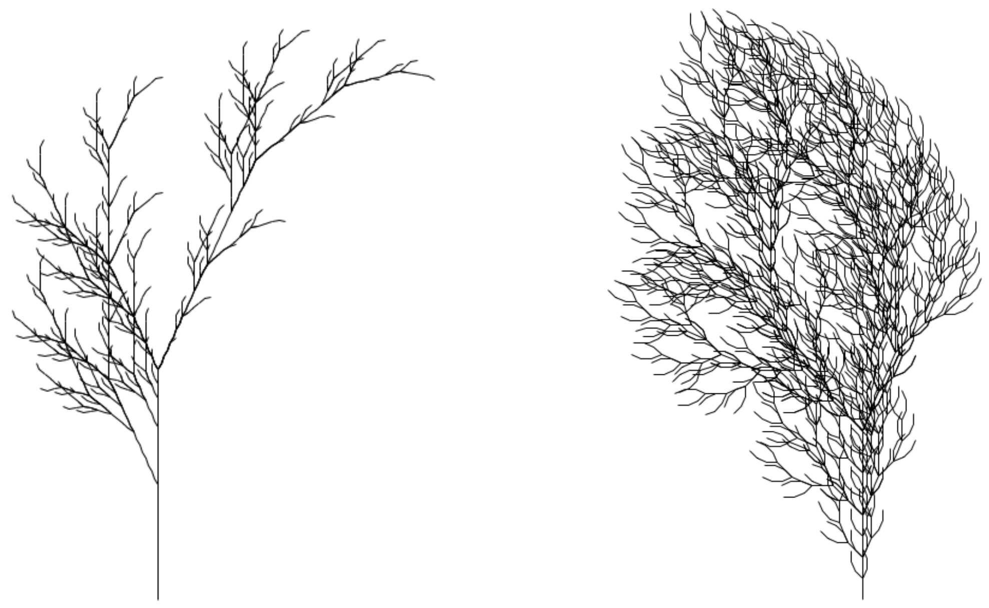


Figure 2 Two trees from *The Algorithmic Beauty of Plants* ([52], p. 25).

Part 1: Draw L-systems with a stack

If you have not completed Exercises 9.6.1 and 9.6.3, do that first. Then incorporate these functions, with the `derive` from Section 9.6, into a complete program. Your `main` function should call the `lsystem` function to draw a particular L-system.

Next, augment the `drawLSystem` function from Exercise 9.6.1 so that it correctly draws L-system strings containing the `[` and `]` characters. Do this by incorporating a single stack into your function, as we described above. Test your function with the three tree-like L-systems above.

Part 2: Draw L-systems recursively

The `drawLSystem` function can be implemented without an explicit stack by using recursion. Think of the `drawLSystem` function as drawing the figure corresponding to a string situated inside matching square brackets. We will pass the index of the first character after the left square bracket (`[`) as an additional parameter:

```
drawLSystem(tortoise, string, startIndex, angle, distance)
```

The function will return the index of the matching right square bracket (`]`). (We can pretend that there are imaginary square brackets around the entire string for the initial call of the function, so we initially pass in 0 for `startIndex`.) The recursive function will iterate over the indices of the characters in `string`, starting at `startIndex`. (Use a `while` loop, for reasons we will see shortly.) When it encounters a non-bracket character, it should do the same thing it did earlier. When the function encounters a left bracket, it will save the turtle's current position and heading, and then recursively call the function with `startIndex` assigned to the index of the character after the left bracket. When this recursive call returns, the current index should be set to the index returned by the recursive call, and the function should reset the turtle's position and heading to the saved values. When it encounters a right bracket, the function will return the index of the right bracket.

For example, the string below would be processed left to right but when the first left bracket is encountered, the function would be called recursively with index 5 passed in for `startIndex`.

$$\overset{0}{\texttt{[}}\texttt{FFFF[}\underbrace{\overset{5}{\texttt{-}}\texttt{FF[-F[-X]+X]+F[-X]+X}}_{\texttt{drawLSystem(..., 5, ...)}}\overset{26}{\texttt{]}}\texttt{+FF[}\underbrace{\texttt{-F[-X]+X}}\texttt{]+F[}\underbrace{\texttt{-X}}\texttt{]+X]}$$

This recursive call will return 26, the index of the corresponding right bracket, and the `+` symbol at index 27 would be the next character processed in the loop. The function will later make two more recursive calls, marked with the two additional braces above.

Using this description, rewrite `drawLSystem` as a recursive function that does not use an explicit stack. Test your recursive function with the same tree-like L-systems, as above.

Question 9.1.3 *Why can the stack used in Part 1 be replaced by recursion in Part 2? Referring back to Figures 9.11 and 9.12, how are recursive function calls similar to pushing and popping from a stack?*

Question 9.1.4 *Use your program to draw the following additional Lindenmayer systems. For each one, set distance = 5, position = $(0, -300)$, heading = 90, and depth = 6.*

Axiom:	`X`
Productions:	`X` → `F[+X]F[-X]+X`
	`F` → `FF`
Angle:	30 degrees

Axiom:	`H`
Productions:	`H` → `HFX[+H][-H]`
	`X` → `X[-FFF][+FFF]FX`
Angle:	25.7 degrees