*9.6 LINDENMAYER SYSTEMS

Aristid Lindenmayer was a Hungarian biologist who, in 1968, invented an elegant mathematical system, now called a ***Lindenmayer system***, for describing the growth of plants and other multicellular organisms. In this section, we will explore basic Lindenmayer systems; Project 9.1 guides you through the process of creating more complex Lindenmayer systems that can produce realistic two-dimensional images of plants, trees, and bushes.

Formal grammars

A Lindenmayer system is a particular type of ***formal grammar***. A formal grammar defines a set of *productions* (or *rules*) for constructing strings of characters. The following simple grammar defines three productions that allow for the construction of a handful of English sentences.

$S \rightarrow N\ V$

$N \rightarrow$ our dog | the school bus | my foot

$V \rightarrow$ ate my homework | swallowed a fly | barked

The first production, $S \rightarrow N\ V$ says that the symbol S (a special *start symbol*) may be replaced by the string $N\ V$. The second production states that the symbol N (short for "noun phrase") may be replaced by one of three strings: "our dog," "the school bus," or "my foot" (the vertical bar (|) means "or"). The third production states that the symbol V (short for "verb phrase") may be replaced by one of three other strings. The following sequence represents one way to use these productions to derive a sentence.

$$S \Rightarrow N\ V \Rightarrow \text{my foot } V \Rightarrow \text{my foot swallowed a fly}$$

The derivation starts with the start symbol S. Using the first production, S is replaced with the string $N\ V$. Then, using the second production, N is replaced with the string "my foot." Finally, using the third production, V is replaced with "swallowed a fly."

Formal grammars were invented by linguist Noam Chomsky in the 1950's as a model for understanding the common characteristics of human language. Formal grammars are used extensively in computer science to both describe the syntax of programming languages and as formal models of computation. As a model of computation, a grammar's productions represent the kinds of operations that are possible, and the resulting strings, formally called *words*, represent the range of possible outputs. The most general formal grammars, called *unrestricted grammars* are computationally equivalent to Turing machines. Putting restrictions on the types of productions that are allowed in grammars affects their computational power. The standard hierarchy of grammar types is now called the *Chomsky hierarchy*.

L-systems

A Lindenmayer system (or L-system) is a special type of grammar in which

(a) all applicable productions are applied in parallel at each step, and

(b) some of the symbols represent turtle graphics drawing commands.

The parallelism is meant to mimic the parallel nature of cellular division in plants and other multicellular organisms. The turtle graphics commands represented by the symbols in a derived string can be used to draw the growing organism.

Instead of a start symbol, an L-system specifies an *axiom* where all derivations begin. For example, the following grammar is a simple L-system:

Axiom:	`F`
Production:	`F` → `F-F++F-F`

We can see from the following derivation that parallel application of the single production very quickly leads to very long strings:

```
F ⇒ F-F++F-F
  ⇒ F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F
  ⇒ F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F
    ++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++
    F-F-F-F++F-F
  ⇒ F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F
    ++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++
    F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F+
    +F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-
    F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F+
    +F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F
    -F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-
    F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-
    F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F
  ⇒ ···
```

In the first step of the derivation, the production is applied to replace `F` with `F-F++F-F`. In the second step, all four instances of `F` are replaced with `F-F++F-F`. The same process occurs in the third step, and the resulting string grows very quickly. The number of strings generated from the axiom in a derivation is called the *depth* of the derivation. If we stopped the above derivation after the last string shown, then its depth would be four because four strings were generated.

Reflection 1 *How can the derivation algorithm be framed as a recursive algorithm?*

The derivation algorithm has essentially two steps: apply the productions to the current string to get a new string, and then apply the same derivation algorithm to the new string but with depth decreased by one. For example, in the derivation above, applying a derivation with depth four to the axiom `F` is the same as applying a derivation with depth three to `F-F++F-F`, which is the same as applying a derivation of length two to the next string, etc. In general, applying a derivation with depth d to a string is the same as applying a derivation with depth $d - 1$ to that string after the productions have been applied one time. As a recursive algorithm in pseudocode, this process looks like this:

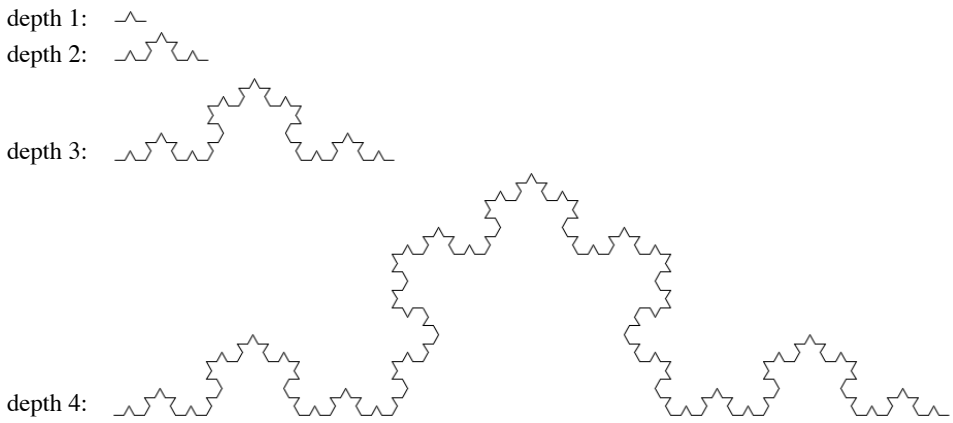


Figure 1 Koch curves resulting from a Lindenmayer system.

Algorithm DERIVE

Input: a *string*, a set of *productions*, and a *depth*

```
1  if depth ≤ 0:
2      return string
3  newString ← result of productions applied once to string
4  return DERIVE (newString, productions, depth − 1)
```

The parameter *depth* controls how many times we apply the productions. The *depth* is decremented in each recursive call, precisely the way we did with fractals in Section 9.1. When *depth* is 0, we do not want to apply the productions at all, so we return the string untouched.

Each symbol in an L-system represents a turtle graphics command:

- `F` means "move forward"
- `-` means "turn left"
- `+` means "turn right"

Interpreted in this way, every derived string represents a sequence of instructions for a turtle to follow. The distance moved for an `F` symbol can be chosen when the string is drawn. But the angle that the turtle turns when it encounters a `-` or `+` symbol must be specified by the L-system. For the L-system above, we will specify an angle of 60 degrees:

Axiom:	`F`
Production:	`F` → `F-F++F-F`
Angle:	60 degrees

Reflection 2 *Carefully follow the turtle graphics instructions (on graph paper) in each of the first two strings derived from this L-system (*`F-F++F-F` *and* `F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F`*). Do the pictures look familiar?*

An annotated sketch of the shorter string is shown below.

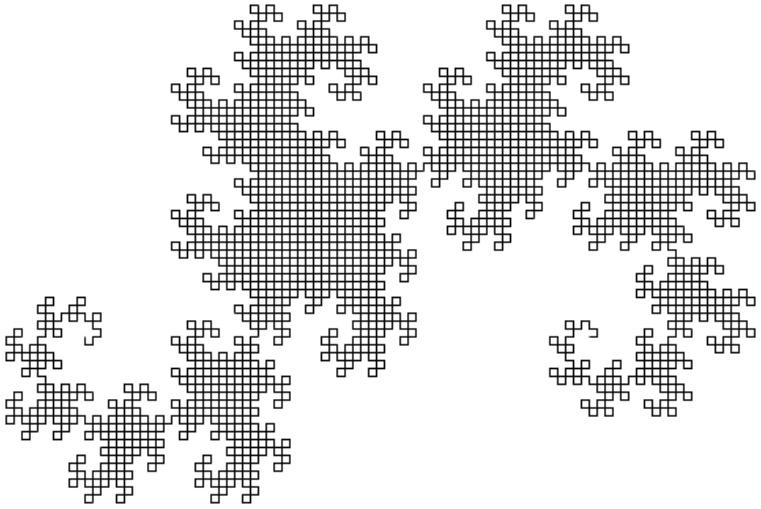
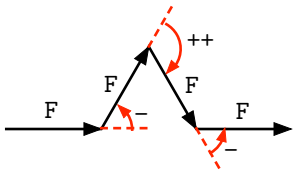


Figure 2 A dragon curve resulting from a Lindenmayer system.



Starting on the left, we first move forward. Then we turn left 60 degrees and move forward again. Next, we turn right twice, a total of 120 degrees. Finally, we move forward, turn left again 60 degrees, and move forward one last time. As shown in Figure 1, the strings derived from this L-system produce Koch curves. Indeed, Lindenmayer systems produce fractals!

Here is another example:

Axiom:	`FX`
Productions:	`X` → `X-YF`
	`Y` → `FX+Y`
Angle:	90 degrees

This L-system produces a well-known fractal known as a dragon curve, shown in Figure 2.

Implementing L-systems

To implement Lindenmayer systems in Python, we need to answer three questions:

1. How do we represent the axiom and productions?
2. How do we apply productions to generate strings?
3. How do we draw the sequence of turtle graphics commands in an L-system string?

Clearly, the axiom and subsequent strings generated by an L-system can be stored as Python strings. The productions can conveniently be stored in a dictionary. For each production, we create an entry in the dictionary with key equal to the symbol on the lefthand side and

value equal to the string on the righthand side. For example, the productions for the dragon curve L-system would be stored as the following dictionary:

```
{'X': 'X-YF', 'Y':'FX+Y'}
```

Once we have the productions in a dictionary, applying them to a string (line 3 in the algorithm) is relatively easy. We iterate over the string one character at a time. For each character, we check if that character is a key in the production dictionary. If it is, we apply the associated production by appending the value associated with that key to the end of a new string. If the character is not in the dictionary, then we simply append the same character to the end of the new string. The following code accomplishes this:

```
newString = ''
for symbol in string:
    if symbol in productions:
        newString = newString + productions[symbol]
    else:
        newString = newString + symbol
```

This loop is incorporated into the following complete implementation of the derivation process in Python.

```
def derive(string, productions, depth):
    """Recursively apply productions to axiom 'depth' times.

    Parameters:
        string:      a string of L-system symbols
        productions: a dictionary containing L-system productions
        depth:       the number of times the productions are applied

    Return value: new string reflecting the application of productions
    """

    if depth <= 0:              # base case
        return string

    newString = ''              # apply productions once to the string
    for symbol in string:
        if symbol in productions:
            newString = newString + productions[symbol]
        else:
            newString = newString + symbol

    return derive(newString, productions, depth - 1)  # recursive call
```

The following `main` function derives a string for the Koch curve with depth 3.

```
def main():
    kochProductions = {'F': 'F-F++F-F'}
    result = derive('F', kochProductions, 3)
    print(result)

main()
```

Reflection 3 *Run the program above. Then modify the* `main` *function so that it derives the depth 4 string for the dragon curve.*

Of course, Lindenmayer systems are much more satisfying when you can draw them. We will leave that to you as an exercise. In Project 9.1, we explore how to augment L-systems so they can produce branching shapes that closely resemble real plants.

Exercises

9.6.1* Write a function

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

that draws the picture described by the given L-system `string`. Your function should correctly handle the special symbols we discussed in this section (`F`, `+`, `-`). Any other symbols should be ignored. The parameters `angle` and `distance` give the angle the turtle turns in response to a `+` or `-` command, and the distance the turtle draws in response to an `F` command, respectively. For example, the following program should draw the smallest Koch curve.

```
def main():
    george = turtle.Turtle()
    george.hideturtle()
    drawLSystem(george, 'F-F++F-F', 60, 10)

main()
```

9.6.2. Apply your `drawLSystem` function from Exercise 9.6.1 to each of the following strings:

(a) `F-F++F-F++F-F++F-F++F-F++F-F` (angle = 60 degrees, distance = 20)

(b) `FX-YF-FX+YF-FX-YF+FX+YF-FX-YF-FX+YF+FX-YF+FX+YF`
(angle = 90 degrees, distance = 20)

9.6.3. Write a function

```
lsystem(axiom, productions, depth, angle, distance, position, heading)
```

that calls the `derive` function with the first three parameters, and then calls your `drawLSystem` function from Exercise 9.6.1 with the new string and the values of `angle` and `distance`. The last two parameters specify the initial `position` and `heading` of the turtle, before `drawLSystem` is called. This function combines all of your previous work into a single L-system generator.

9.6.4. Call your `lsystem` function from Exercise 9.6.3 on each the following L-systems:

(a)

Axiom:	`F`
Production:	`F` → `F-F++F-F`
Angle:	60 degrees

distance = 10, position = (−400,0), heading = 0, depth = 4

(b)

Axiom:	`FX`
Productions:	`X` → `X-YF`
	`Y` → `FX+Y`
Angle:	90 degrees

distance = 5, position = (0,0), heading = 0, depth = 12

(c)

Axiom:	`F-F-F-F`
Production:	`F` → `F-F+F+FF-F-F+F`
Angle:	90 degrees

distance = 3, position = (−100, − 100), heading = 0, depth = 3

(d)

Axiom:	`F-F-F-F`
Production:	`F` → `FF-F-F-F-F-F+F`
Angle:	90 degrees

distance = 5, position = (0, − 200), heading = 0, depth = 3

9.6.5. By simply changing the axiom, we can turn the L-system for the Koch curve discussed in the text an L-system for a Koch snowflake composed of three Koch curves. Show what the axiom needs to be. Use your `lsystem` function from Exercise 9.6.3 to work out and test your answer.

Selected Exercise Solutions

9.6.1
```
def drawLSystem(tortoise, string, angle, distance):
    for symbol in string:
        if symbol == 'F':
            george.forward(distance)
        elif symbol == '+':
            george.right(angle)
        elif symbol == '-':
            george.left(angle)
```