16. Recursion

16.1. What Is Recursion?

16.2. Calculating the Sum of a List of Numbers

16.3. The Three Laws of Recursion

16.4. Converting an Integer to a String in Any Base

16.5. Visualizing Recursion

16.6. Sierpinski Triangle

16.7. Glossary

16.8. Programming Exercises

16.9. Exercises

16.1. What Is Recursion?

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

16.2. Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1,3,5,7,9]. An iterative function that computes the sum is shown below. The function uses an accumulator variable (theSum) to compute a running total of all the numbers in the list by starting with 0 and adding each number in the list.

1 def listsum(numList):

2 theSum = 0

3 or i in numList:

4 theSum = theSum + i

5 return theSum

6

​7 print(listsum([1,3,5,7,9]))

8

​

Pretend for a minute that you do not have while loops or for loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:

((((1+3)+5)+7)+9)

We can also parenthesize the expression the other way around,

(1+(3+(5+(7+9))))

Notice that the innermost set of parentheses, (7+9), is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

total= (1+(3+(5+(7+9))))total= (1+(3+(5+16)))total= (1+(3+21))total= (1+24)total= 25

How can we take this idea and turn it into a Python program? First, let’s restate the sum problem in terms of Python lists. We might say the the sum of the list numList is the sum of the first element of the list (numList[0]), and the sum of the numbers in the rest of the list (numList[1:]). To state it in a functional form:

listSum(numList)=first(numList)+listSum(rest(numList))

In this equation first(numList) returns the first element of the list and rest(numList) returns a list of everything but the first element. This is easily expressed in Python.

1 def listsum(numList):

2 if len(numList) == 1:

3 return numList[0]

4 else:

5 return numList[0] + listsum(numList[1:])

6

​7 print(listsum([1,3,5,7,9]))

8

There are a few key ideas in this listing to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls itself! This is the reason that we call the listsum algorithm recursive. A recursive function is a function that calls itself.

Figure 1 shows the series of recursive calls that are needed to sum the list [1,3,5,7,9]. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 2 shows the additions that are performed as listsum works its way backward through the series of calls. When listsum returns from the topmost problem, we have the solution to the whole problem.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

16.2. Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1,3,5,7,9]. An iterative function that computes the sum is shown below. The function uses an accumulator variable (theSum) to compute a running total of all the numbers in the list by starting with 0 and adding each number in the list.

1

def listsum(numList):

2

theSum = 0

3

for i in numList:

4

theSum = theSum + i

5

return theSum

6

​

7

print(listsum([1,3,5,7,9]))

8

​

Activity: 16.2.1 Iterative Summation (lst\_itsum)

Pretend for a minute that you do not have while loops or for loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:

((((1+3)+5)+7)+9)

We can also parenthesize the expression the other way around,

(1+(3+(5+(7+9))))

Notice that the innermost set of parentheses, (7+9), is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

total= (1+(3+(5+(7+9))))total= (1+(3+(5+16)))total= (1+(3+21))total= (1+24)total= 25

How can we take this idea and turn it into a Python program? First, let’s restate the sum problem in terms of Python lists. We might say the the sum of the list numList is the sum of the first element of the list (numList[0]), and the sum of the numbers in the rest of the list (numList[1:]). To state it in a functional form:

listSum(numList)=first(numList)+listSum(rest(numList))

In this equation first(numList) returns the first element of the list and rest(numList) returns a list of everything but the first element. This is easily expressed in Python.

1 def listsum(numList):

2 if len(numList) == 1:

3 return numList[0]

4 else:

5 return numList[0] + listsum(numList[1:])

6

​7 print(listsum([1,3,5,7,9]))

8

​

There are a few key ideas in this listing to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls itself! This is the reason that we call the listsum algorithm recursive. A recursive function is a function that calls itself.

Figure 1 shows the series of recursive calls that are needed to sum the list [1,3,5,7,9]. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 2 shows the additions that are performed as listsum works its way backward through the series of calls. When listsum returns from the topmost problem, we have the solution to the whole problem.

This Chapter

16.2. Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1,3,5,7,9]. An iterative function that computes the sum is shown below. The function uses an accumulator variable (theSum) to compute a running total of all the numbers in the list by starting with 0 and adding each number in the list.

1 def listsum(numList):

2 theSum = 0

3 for i in numList:

4 TheSum = theSum + i

5 return theSum

6

​7 print(listsum([1,3,5,7,9]))

8

Pretend for a minute that you do not have while loops or for loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:

((((1+3)+5)+7)+9)

We can also parenthesize the expression the other way around,

(1+(3+(5+(7+9))))

Notice that the innermost set of parentheses, (7+9), is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

total= (1+(3+(5+(7+9))))total= (1+(3+(5+16)))total= (1+(3+21))total= (1+24)total= 25

How can we take this idea and turn it into a Python program? First, let’s restate the sum problem in terms of Python lists. We might say the the sum of the list numList is the sum of the first element of the list (numList[0]), and the sum of the numbers in the rest of the list (numList[1:]). To state it in a functional form:

listSum(numList)=first(numList)+listSum(rest(numList))

In this equation first(numList) returns the first element of the list and rest(numList) returns a list of everything but the first element. This is easily expressed in Python.

1 def listsum(numList):

2 If len(numList) == 1:

3 return numList[0]

4 else:

5 return numList[0] + listsum(numList[1:])

6

​7 print(listsum([1,3,5,7,9]))

8

​

There are a few key ideas in this listing to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls itself! This is the reason that we call the listsum algorithm recursive. A recursive function is a function that calls itself.

Figure 1 shows the series of recursive calls that are needed to sum the list [1,3,5,7,9]. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 2 shows the additions that are performed as listsum works its way backward through the series of calls. When listsum returns from the topmost problem, we have the solution to the whole problem.

16.2. Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1,3,5,7,9]. An iterative function that computes the sum is shown below. The function uses an accumulator variable (theSum) to compute a running total of all the numbers in the list by starting with 0 and adding each number in the list.

1 def listsum(numList):

2 theSum = 0

3 or i in numList:

4 theSum = theSum + i

5 return theSum

6

​7 print(listsum([1,3,5,7,9]))

8

​

Activity: 16.2.1 Iterative Summation (lst\_itsum)

Pretend for a minute that you do not have while loops or for loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:

((((1+3)+5)+7)+9)

We can also parenthesize the expression the other way around,

(1+(3+(5+(7+9))))

Notice that the innermost set of parentheses, (7+9), is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

total= (1+(3+(5+(7+9))))total= (1+(3+(5+16)))total= (1+(3+21))total= (1+24)total= 25

How can we take this idea and turn it into a Python program? First, let’s restate the sum problem in terms of Python lists. We might say the the sum of the list numList is the sum of the first element of the list (numList[0]), and the sum of the numbers in the rest of the list (numList[1:]). To state it in a functional form:

listSum(numList)=first(numList)+listSum(rest(numList))

In this equation first(numList) returns the first element of the list and rest(numList) returns a list of everything but the first element. This is easily expressed in Python.

1 def listsum(numList):

2 if len(numList) == 1:

3 return numList[0]

4 else:

5 r eturn numList[0] + listsum(numList[1:])

6

​7 print(listsum([1,3,5,7,9]))

8

​

Activity: 16.2.2 Recursive Summation (lst\_recsum)

There are a few key ideas in this listing to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls itself! This is the reason that we call the listsum algorithm recursive. A recursive function is a function that calls itself.

Figure 1 shows the series of recursive calls that are needed to sum the list [1,3,5,7,9]. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 2 shows the additions that are performed as listsum works its way backward through the series of calls. When listsum returns from the topmost problem, we have the solution to the whole problem.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

16.6. Sierpinski Triangle

Another fractal that exhibits the property of self-similarity is the Sierpinski triangle. An example is shown in Figure 3. The Sierpinski triangle illustrates a three-way recursive algorithm. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into four new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles. You can continue to apply this procedure indefinitely if you have a sharp enough pencil. Before you continue reading, you may want to try drawing the Sierpinski triangle yourself, using the method described.

../\_images/sierpinski.png

Figure 3: The Sierpinski Triangle

Since we can continue to apply the algorithm indefinitely, what is the base case? We will see that the base case is set arbitrarily as the number of times we want to divide the triangle into pieces. Sometimes we call this number the “degree” of the fractal. Each time we make a recursive call, we subtract 1 from the degree until we reach 0. When we reach a degree of 0, we stop making recursive calls. The code that generated this Sierpinski Triangle is shown below.

1

import turtle

2

​

3 def drawTriangle(points,color,myTurtle):

4 myTurtle.fillcolor(color)

5 myTurtle.up()

6 myTurtle.goto(points[0][0],points[0][1])

7 myTurtle.down()

8 myTurtle.begin\_fill()

9 myTurtle.goto(points[1][0],points[1][1])

10 myTurtle.goto(points[2][0],points[2][1])

11 myTurtle.goto(points[0][0],points[0][1])

12 myTurtle.end\_fill()

13

​14 def getMid(p1,p2):

15 return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

16

​17 def sierpinski(points,degree,myTurtle):

18 colormap = ['blue','red','green','white','yellow',

19 violet','orange']

20 drawTriangle(points,colormap[degree],myTurtle)

21 iif degree > 0:

22 sierpinski([points[0],

23 getMid(points[0], points[1]),

24 getMid(points[0], points[2])],

This program follows the ideas outlined above. The first thing sierpinski does is draw the outer triangle. Next, there are three recursive calls, one for each of the new corner triangles we get when we connect the midpoints.

Look at the code and think about the order in which the triangles will be drawn. While the exact order of the corners depends upon how the initial set is specified, let’s assume that the corners are ordered lower left, top, lower right. Because of the way the sierpinski function calls itself, sierpinski works its way to the smallest allowed triangle in the lower-left corner, and then begins to fill out the rest of the triangles working back. Then it fills in the triangles in the top corner by working toward the smallest, topmost triangle. Finally, it fills in the lower-right corner, working its way toward the smallest triangle in the lower right.

Sometimes it is helpful to think of a recursive algorithm in terms of a diagram of function calls. Figure 4 shows that the recursive calls are always made going to the left. The active functions are outlined in black, and the inactive function calls are in gray. The farther you go toward the bottom of Figure 4, the smaller the triangles. The function finishes drawing one level at a time; once it is finished with the bottom left it moves to the bottom middle, and so on.

The sierpinski function relies heavily on the getMid function. getMid takes as arguments two endpoints and returns the point halfway between them. In addition, this program has a function that draws a filled triangle using the begin\_fill and end\_fill turtle methods.

16.7. Glossary

base case

A branch of the conditional statement in a recursive function that does not give rise to further recursive calls.

data structure

An organization of data for the purpose of making it easier to use.

immutable data type

A data type which cannot be modified. Assignments to elements or slices of immutable types cause a runtime error.

infinite recursion

A function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.

mutable data type

A data type which can be modified. All mutable types are compound types. Lists and dictionaries (see next chapter) are mutable data types; strings and tuples are not.

recursion

The process of calling the function that is already executing.

recursive call

The statement that calls an already executing function. Recursion can even be indirect — function f can call g which calls h, and h could make a call back to f.

recursive definition

A definition which defines something in terms of itself. To be useful it must include base cases which are not recursive. In this way it differs from a circular definition. Recursive definitions often provide an elegant way to express complex data structures.

tuple

A data type that contains a sequence of elements of any type, like a list, but is immutable. Tuples can be used wherever an immutable type is required, such as a key in a dictionary (see next chapter).

tuple assignment

An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs in parallel rather than in sequence, making it useful for swapping values.

This Chapter

16.8. Programming Exercises

Write a recursive function to compute the factorial of a number.

Write a recursive function to reverse a list.

Modify the recursive tree program using one or all of the following ideas:

Modify the thickness of the branches so that as the branchLen gets smaller, the line gets thinner.

Modify the color of the branches so that as the branchLen gets very short it is colored like a leaf.

Modify the angle used in turning the turtle so that at each branch point the angle is selected at random in some range. For example choose the angle between 15 and 45 degrees. Play around to see what looks good.

Modify the branchLen recursively so that instead of always subtracting the same amount you subtract a random amount in some range.

If you implement all of the above ideas you will have a very realistic looking tree.

Find or invent an algorithm for drawing a fractal mountain. Hint: One approach to this uses triangles again.

Write a recursive function to compute the Fibonacci sequence. How does the performance of the recursive function compare to that of an iterative version?

Implement a solution to the Tower of Hanoi using three stacks to keep track of the disks.

Using the turtle graphics module, write a recursive program to display a Hilbert curve.

Using the turtle graphics module, write a recursive program to display a Koch snowflake.

Write a program to solve the following problem: You have two jugs: a 4-gallon jug and a 3-gallon jug. Neither of the jugs have markings on them. There is a pump that can be used to fill the jugs with water. How can you get exactly two gallons of water in the 4-gallon jug?

Generalize the problem above so that the parameters to your solution include the sizes of each jug and the final amount of water to be left in the larger jug.

Write a program that solves the following problem: Three missionaries and three cannibals come to a river and find a boat that holds two people. Everyone must get across the river to continue on the journey. However, if the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. Find a series of crossings that will get everyone safely to the other side of the river.

Modify the Tower of Hanoi program using turtle graphics to animate the movement of the disks. Hint: You can make multiple turtles and have them shaped like rectangles.

Pascal’s triangle is a number triangle with numbers arranged in staggered rows such that

anr=n!r!(n−r)!

This equation is the equation for a binomial coefficient. You can build Pascal’s triangle by adding the two numbers that are diagonally above a number in the triangle. An example of Pascal’s triangle is shown below.

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Write a program that prints out Pascal’s triangle. Your program should accept a parameter that tells how many rows of the triangle to print.