Recursion 8

In Chap. 4 we explored many aspects of functions, including functions that call other functions. A closely related topic that we did not consider previously is whether or not a function can call itself. It turns out that this is, in fact, possible, and that it is a powerful technique for solving some problems.

A definition that describes something in terms of itself is *recursive*. To be useful a recursive definition must describe whatever is being defined in terms of a different (typically a smaller or simpler) version of itself. A definition that defines something in terms of the same version of itself, while recursive, is not particularly useful because the definition is circular. A useful recursive definition must make progress toward a version of the problem with a known solution.

Any function that calls itself is recursive because the function's body (its definition) includes a call to the function that is being defined. In order to reach a solution a recursive function must have at least one case where it is able to produce the required result without calling itself. This is referred to as the *base case*. Cases where the function calls itself are referred to as *recursive cases*. Our examination of recursion will continue with three examples.

8.1 Summing Integers

Consider the problem of computing the sum of all the integers from 0 up to and including some positive integer, `n`. This can be accomplished using a loop or a formula. It can also be performed recursively. The simplest case is when `n` is 0. In this case the answer is known to be 0 and that answer can be returned without using another version of the problem. As a result, this is the base case.

For any positive integer, `n`, the sum of the values from 0 up to and including `n` can be computed by adding `n` to the sum of the numbers from 0 up to and including `n - 1`. This description is recursive because the sum of `n` integers is expressed as a smaller version of the same problem (summing the numbers from 0 up to and including `n - 1`), plus a small amount of additional work (adding `n` to that sum). Each time this recursive definition is applied it makes progress toward the base case (when `n` is 0). When the base case is reached no further recursion is performed. This allows the calculation to complete and the result to be returned.

The following program implements the recursive algorithm described in the previous paragraphs to compute the sum of the integers from 0 up to and including a positive integer entered by the user. An if statement is used to determine whether to execute the base case or the recursive case. When the base case executes 0 is returned immediately, without making a recursive call. When the recursive case executes the function is called again with a smaller argument (`n - 1`). Once the recursive call returns, `n` is added to the returned value. This successfully computes the total of the values from 0 up to and including `n`. Then this total is returned as the function's result.

```
# Compute the sum of the integers from 0 up to and including n using recursion
# @param n the maximum value to include in the sum
# @return the sum of the integers from 0 up to and including n
def sum_to(n):
  if n <= 0:
    return 0                    # Base case
  else:
    return n + sum_to(n - 1)    # Recursive case

# Compute the sum of the integers from 0 up to and including a value entered by the user
num = int(input("Enter a non-negative integer: "))
total = sum_to(num)
print("The total of the integers from 0 up to and including", \
      num, "is", total)
```

Consider what happens if the user enters 2 when the program is run. This value is read from the keyboard and converted into an integer. Then `sum_to` is called with 2 as its argument. The `if` statement's condition evaluates to `True` so its body executes and `sum_to` is called recursively with its argument equal to 1. This recursive call must complete before the the copy of `sum_to` where `n` is equal to 2 can compute and return its result.

Executing the recursive call to `sum_to` where `n` is equal to 1 causes another recursive call to `sum_to` with `n` equal to 0. Once that recursive call begins to execute there are three copies of `sum_to` executing with argument values 2, 1 and 0. The copy of `sum_to` where `n` is equal to 2 is waiting for the copy where `n` is equal to 1 to complete before it can return its result, and the copy where `n` is equal to 1 is waiting for the copy where `n` is equal to 0 to complete before it can return its result. While all of the functions have the same name, each copy of the function is entirely separate from all of the other copies.

When `sum_to` executes with `n` equal to 0 the base case is executed. It immediately returns 0. This allows the version of `sum_to` where `n` is equal to 1 to progress by adding 1 to the 0 returned by the recursive call. Then the total, which is 1, is returned by the call where `n` is equal to 1. Execution continues with the version of `sum_to` where `n` is equal to 2. It adds `n`, which is 2, to the 1 returned by the recursive call, and 3 is returned and stored into `total`. Finally, the total is displayed and the program terminates.

8.2 Fibonacci Numbers

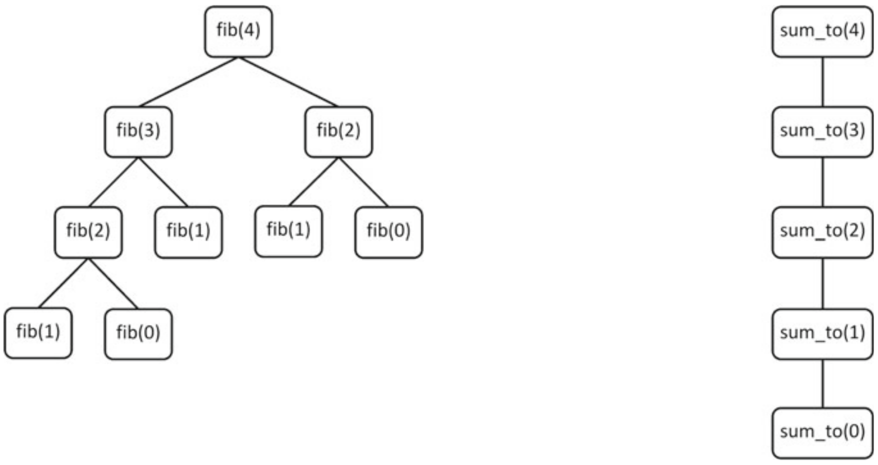
The Fibonacci numbers are a sequence of integers that begin with 0 and 1. Each subsequent number in the sequence is the sum of its two immediate predecessors. As a result, the first 10 numbers in the Fibonacci sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21 and 34. Numbers in the Fibonacci sequence are commonly denoted by F_n, where n is a non-negative integer identifying the number's index within the sequence (starting from 0).

Numbers in the Fibonacci sequence, beyond the first two, can be computed using the formula $F_n = F_{n-1} + F_{n-2}$. This definition is recursive because a larger Fibonacci number is computed using two smaller Fibonacci numbers. The first two numbers in the sequence, F_0 and F_1, are the base cases because they have known values that are not computed recursively. A program that implements this recursive formula for computing Fibonacci numbers is shown below. It computes and displays the value of F_n for some value of n entered by the user.

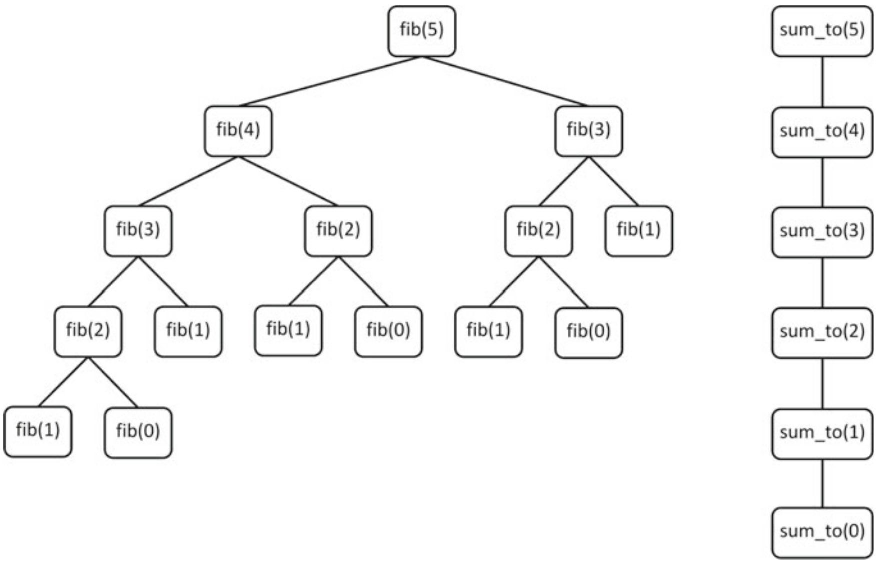
```python
# Compute the nth Fibonacci number using recursion
# @param n the index of the Fibonacci number to compute
# @return the nth Fibonacci number
def fib(n):
  # Base cases
  if n == 0:
    return 0
  if n == 1:
    return 1

  # Recursive case
  return fib(n-1) + fib(n-2)

# Compute the Fibonacci number requested by the user
n = int(input("Enter a non-negative integer: "))
print("fib(%d) is %d." % (n, fib(n)))
```

This recursive algorithm for computing Fibonacci numbers is compact, but it is slow, even when working with fairly modest values. While computing `fib(35)` will return quickly on a modern machine, computing `fib(70)` will take years to complete. As a result, larger Fibonacci numbers are normally computed using a loop or a formula.

Based on the performance of our Fibonacci numbers program you might be tempted to conclude that recursive solutions are too slow to be useful. While that is true in this particular situation, it is not true in general. Our previous program that



(a) The Function Calls Used to Compute `fib(4)` and `sum_to(4)`



(b) The Function Calls Used to Compute `fib(5)` and `sum_to(5)`

Fig. 8.1 A comparison of the function calls for `fib` and `sum_to`

summed integers ran quickly even for larger values, and there are some problems that have very efficient recursive algorithms, such as Euclid's algorithm for computing the greatest common divisor of two integers which is described in Exercise 174.

Figure 8.1 illustrates the recursive calls made when computing F_4 and F_5, as well as the recursive calls made to evaluate `sum_to(4)` and `sum_to(5)`. Comparing the function calls made to compute these results for different input values illustrates the difference in efficiency for these problems.

When the argument passed to sum_to increases from 4 to 5 the number of function calls also increases from 4 to 5. More generally, when the argument passed to sum_to increases by 1 the number of function calls also increases by 1. This is referred to as linear growth because the number of recursive calls is directly proportional to the value of the argument provided when the function is first called.

In contrast, when the argument passed to fib increases from 4 to 5 the number of function calls increases from 9 to 15. More generally, when the position of the Fibonacci number being computed increases by 1, the number of recursive calls (nearly) doubles. This is referred to as exponential growth. Exponential growth makes it impossible (in any practical sense) to calculate the result for large values because repeatedly doubling the time needed for the computation quickly results in a running time that is simply too long to be useful.

8.3 Counting Characters

Recursion can be used to solve any problem that can be expressed in terms of itself. It is not restricted to problems that operate on integers. For example, consider the problem of counting the number of occurrences of a particular character, ch, within a string, s. A recursive function for solving this problem can be written that takes s and ch as arguments and returns the number of times that ch occurs in s as its result.

The base case for this problem is s being the empty string. Since an empty string does not contain any characters it must contain 0 occurrences of ch. As a result, the function can return 0 in this case without making a recursive call.

The number of occurrences of ch in a longer string can be determined in the following recursive manner. To help simplify the description of the recursive case we will define the tail of s to be all of the characters in s except for the first character. The tail of a string containing only one character is the empty string.

If the first character in s is ch then the number of occurrences of ch in s is one plus the number of occurrences of ch in the tail of s. Otherwise the number of occurrences of ch in s is the number of occurrences of ch in the tail of s. This definition makes progress toward the base case (when s is the empty string) because the tail of s is always shorter than s. A program that implements this recursive algorithm is shown below.

```
# Count the number of times a particular character is present in a string
# @param s the string in which the characters are counted
# @param ch the character to count
# @return the number of occurrences of ch in s
def count(s, ch):
  if s == "":
    return 0   # Base case

  # Compute the tail of s
  tail = s[1 : len(s)]
  # Recursive cases
  if ch == s[0]:
    return 1 + count(tail, ch)
  else:
    return count(tail, ch)

# Count the number of times a character entered by the user occurs in a string entered
# by the user
s = input("Enter a string: ")
ch = input("Enter the character to count: ")
print("'%s' occurs %d times in '%s'" % (ch, count(s, ch), s))
```

8.4 Exercises

A recursive function is a function that calls itself. Such functions normally include one or more base cases and one or more recursive cases. When the base case executes the result for the function is computed without making a recursive call. Recursive cases compute their result by making one or more recursive calls, typically to a smaller or simpler version of the problem. All of the exercises in this chapter should be solved by writing one or more recursive functions. Each of these functions will call itself, and may also make use of any of the Python features that were discussed in the previous chapters.

Exercise 173: Total the Values

(*Solved, 29 Lines*)

Write a program that reads values from the user until a blank line is entered. Display the total of all of the values entered by the user (or 0.0 if the first value entered is a blank line). Complete this task using recursion. Your program may not use any loops.

> Hint: The body of your recursive function will need to read one value from the user, and then determine whether or not to make a recursive call. Your function does not need to take any arguments, but it will need to return a numeric result.

Exercise 174: Greatest Common Divisor

(*24 Lines*)

Euclid was a Greek mathematician who lived approximately 2,300 years ago. His algorithm for computing the greatest common divisor of two positive integers, `a` and `b`, is both efficient and recursive. It is outlined below:

```
If $b$ is 0 then
    Return $a$
Else
    Set $c$ equal to the remainder when $a$ is divided by $b$
    Return the greatest common divisor of $b$ and $c$
```

Write a program that implements Euclid's algorithm and uses it to determine the greatest common divisor of two integers entered by the user. Test your program with some very large integers. The result will be computed quickly, even for huge numbers consisting of hundreds of digits, because Euclid's algorithm is extremely efficient.

Exercise 175: Recursive Decimal to Binary

(*34 Lines*)

In Exercise 82 you wrote a program that used a loop to convert a decimal number to its binary representation. In this exercise you will perform the same task using recursion.

Write a recursive function that converts a non-negative decimal number to binary. Treat 0 and 1 as base cases which return a string containing the appropriate digit. For all other positive integers, n, you should compute the next digit using the remainder operator and then make a recursive call to compute the digits of n // 2. Finally, you should concatenate the result of the recursive call (which will be a string) and the next digit (which you will need to convert to a string) and return this string as the result of the function.

Write a main program that uses your recursive function to convert a non-negative integer entered by the user from decimal to binary. Your program should display an appropriate error message if the user enters a negative value.

Exercise 176: The NATO Phonetic Alphabet

(*33 Lines*)

A spelling alphabet is a set of words, each of which stands for one of the 26 letters in the alphabet. While many letters are easily misheard over a low quality or noisy communication channel, the words used to represent the letters in a spelling alphabet are generally chosen so that each sounds distinct and is difficult to confuse with any other. The NATO phonetic alphabet is a widely used spelling alphabet. Each letter and its associated word is shown in Table 8.1.

Write a program that reads a word from the user and then displays its phonetic spelling. For example, if the user enters `Hello` then the program should output `Hotel Echo Lima Lima Oscar`. Your program should use a recursive function to perform this task. Do not use a loop anywhere in your solution. Any non-letter characters entered by the user should be ignored.

Table 8.1 NATO phonetic alphabet

Letter	Word	Letter	Word	Letter	Word
A	Alpha	J	Juliet	S	Sierra
B	Bravo	K	Kilo	T	Tango
C	Charlie	L	Lima	U	Uniform
D	Delta	M	Mike	V	Victor
E	Echo	N	November	W	Whiskey
F	Foxtrot	O	Oscar	X	Xray
G	Golf	P	Papa	Y	Yankee
H	Hotel	Q	Quebec	Z	Zulu
I	India	R	Romeo		

Exercise 177: Roman Numerals

(*25 Lines*)

As the name implies, Roman numerals were developed in ancient Rome. Even after the Roman empire fell, its numerals continued to be widely used in Europe until the late middle ages, and its numerals are still used in limited circumstances today.

Roman numerals are constructed from the letters M, D, C, L, X, V and I which represent 1000, 500, 100, 50, 10, 5 and 1 respectively. The numerals are generally written from largest value to smallest value. When this occurs the value of the number is the sum of the values of all of its numerals. If a smaller value precedes a larger value then the smaller value is subtracted from the larger value that it immediately precedes, and that difference is added to the value of the number.[1]

Create a recursive function that converts a Roman numeral to an integer. Your function should process one or two characters at the beginning of the string, and then call itself recursively on all of the unprocessed characters. Use an empty string, which has the value 0, for the base case. In addition, write a main program that reads a Roman numeral from the user and displays its value. You can assume that the value entered by the user is valid. Your program does not need to do any error checking.

Exercise 178: Recursive Palindrome

(*Solved, 30 Lines*)

The notion of a palindrome was introduced previously in Exercise 75. In this exercise you will write a recursive function that determines whether or not a string is a palindrome. The empty string is a palindrome, as is any string containing only one

[1]Only C, X and I are used in a subtractive manner. The numeral that a C, X or I precedes must have a value that is no more than 10 times the value being subtracted. As such, I can precede V or X, but it cannot precede L, C, D or M. This means, for example, that 99 must be represented by XCIX rather than by IC.

character. Any longer string is a palindrome if its first and last characters match, and if the string formed by removing the first and last characters is also a palindrome.

Write a main program that reads a string from the user and uses your recursive function to determine whether or not it is a palindrome. Then your program should display an appropriate message for the user.

Exercise 179: Recursive Square Root

(*20 Lines*)

Exercise 74 explored how iteration can be used to compute the square root of a number. In that exercise a better approximation of the square root was generated with each additional loop iteration. In this exercise you will use the same approximation strategy, but you will use recursion instead of a loop.

Create a square root function with two parameters. The first parameter, n, will be the number for which the square root is being computed. The second parameter, *guess*, will be the current guess for the square root. The *guess* parameter should have a default value of 1.0. Do not provide a default value for the first parameter.

Your square root function will be recursive. The base case occurs when $guess^2$ is within 10^{-12} of n. In this case your function should return $guess$ because it is close enough to the square root of n. Otherwise your function should return the result of calling itself recursively with n as the first parameter and $\frac{guess+\frac{n}{guess}}{2}$ as the second parameter.

Write a main program that demonstrate your square root function by computing the square root of several different values. When you call your square root function from the main program you should only pass one parameter to it so that the default value is used for $guess$.

Exercise 180: String Edit Distance

(*Solved, 43 Lines*)

The edit distance between two strings is a measure of their similarity. The smaller the edit distance, the more similar the strings are with regard to the minimum number of insert, delete and substitute operations needed to transform one string into the other.

Consider the strings `kitten` and `sitting`. The first string can be transformed into the second string with the following operations: Substitute the `k` with an `s`, substitute the `e` with an `i`, and insert a `g` at the end of the string. This is the smallest number of operations that can be performed to transform `kitten` into `sitting`. As a result, the edit distance is 3.

Write a recursive function that computes the edit distance between two strings. Use the following algorithm:

```
Let *s* and *t* be the strings
**If** the length of *s* is 0 **then**
    **Return** the length of *t*
**Else if** the length of *t* is 0 **then**
    **Return** the length of *s*
**Else**
    Set *cost* to 0
    **If** the last character in *s* does not equal the last character in *t* **then**
        Set *cost* to 1
    Set *d1* equal to the edit distance between all characters except the last one
        in *s*, and all characters in *t*, plus 1
    Set *d2* equal to the edit distance between all characters in *s*, and all
        characters except the last one in *t*, plus 1
    Set *d3* equal to the edit distance between all characters except the last one
        in *s*, and all characters except the last one in *t*, plus *cost*
    **Return** the minimum of *d1*, *d2* and *d3*
```

Use your recursive function to write a program that reads two strings from the user and displays the edit distance between them.

Exercise 181: Possible Change

(*41 Lines*)

Create a program that determines whether or not it is possible to construct a particular total using a specific number of coins. For example, it is possible to have a total of \$1.00 using four coins if they are all quarters. However, there is no way to have a total of \$1.00 using 5 coins. Yet it is possible to have \$1.00 using 6 coins by using 3 quarters, 2 dimes and a nickel. Similarly, a total of \$1.25 can be formed using 5 coins or 8 coins, but a total of \$1.25 cannot be formed using 4, 6 or 7 coins.

Your program should read both the dollar amount and the number of coins from the user. Then it should display a clear message indicating whether or not the entered dollar amount can be formed using the number of coins indicated. Assume the existence of quarters, dimes, nickels and pennies when completing this problem. Your solution must use recursion. It cannot contain any loops.

Exercise 182: Spelling with Element Symbols

(*67 Lines*)

Each chemical element has a standard symbol that is one, two or three letters long. One game that some people like to play is to determine whether or not a word can be spelled using only element symbols. For example, silicon can be spelled using the symbols Si, Li, C, O and N. However, hydrogen cannot be spelled with any combination of element symbols.

Write a recursive function that determines whether or not a word can be spelled using only element symbols. Your function will require two parameters: the word that you are trying to spell and a list of the symbols that can be used. It will return a string containing the symbols used to achieve the spelling as its result, or an empty string if no spelling exists. Capitalization should be ignored when your function searches for a spelling.

Create a program that uses your function to find and display all of the element names that can be spelled using only element symbols. Display the names of the elements along with the sequences of symbols. For example, one line of your output will be:

```
Silver can be spelled as SiLvEr
```

Your program will use the elements data set, which can be downloaded from the author's website. This data set includes the names and symbols of all 118 chemical elements.

Exercise 183: Element Sequences

(*Solved, 81 Lines*)

Some people like to play a game that constructs a sequence of chemical elements where each element in the sequence begins with the last letter of its predecessor. For example, if a sequence begins with Hydrogen, then the next element must be an element that begins with N, such as Nickel. The element following Nickel must begin with L, such as Lithium. The element sequence that is constructed cannot contain any duplicates. When played alone the goal of the game is to constructed the longest possible sequence of elements. When played with two players, the goal is to select an element that leaves your opponent without an option to add to the sequence.

Write a program that reads the name of an element from the user and uses a recursive function to find the longest sequence of elements that begins with that value. Display the sequence once it has been computed. Ensure that your program responds in a reasonable way if the user does not enter a valid element name.

Hint: It may take your program up to two minutes to find the longest sequence for some elements. As a result, you might want to use elements like Molybdenum and Magnesium as your first test cases. Each has a longest sequence that is only 8 elements long which your program should find in a fraction of a second.

Exercise 184: Flatten a List

(*Solved, 33 Lines*)

Python's lists can contain other lists. When one list occurs inside another the inner list is said to be nested inside the outer list. Each of the inner lists nested within the outer list may also contain nested lists, and those lists may contain additional nested lists to any depth. For example, the following list includes elements that are nested at several different depths: `[1, [2, 3], [4, [5, [6, 7]]], [[[8], 9], [10]]]`.

Lists that contain multiple levels of nesting can be useful when describing complex relationships between values, but such lists can also make performing some operations on those values difficult because the values are nested at different levels. Flattening a list is the process of converting a list that may contain multiple levels of nested lists into a list that contains all of the same elements without any nesting. For example, flattening the list from the previous paragraph results in `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. The following recursive algorithm can be used to flatten a list named *data*:

```
If data is empty then
    Return the empty list
If the first element in data is a list then
    Set l1 to the result of flattening the first element in data
    Set l2 to the result of flattening all of the elements in data, except the first
    Return the concatenation of l1 and l2
If the first element in data is not a list then
    Set l1 to a list containing only the first element in data
    Set l2 to the result of flattening all of the elements in data, except the first
    Return the concatenation of l1 and l2
```

Write a function that implements the recursive flattening algorithm described previously. Your function will take one argument, which is the list to flatten, and it will return one result, which is the flattened list. Include a main program that demonstrates that your function successfully flattens the list shown earlier in this problem, as well as several others.

Hint: Python includes a function named `type` which returns the type of its only argument. Information about using this function to determine whether or not a variable is a list can be found online.

Exercise 185: Run-Length Decoding

(*33 Lines*)

Run-length encoding is a simple data compression technique that can be effective when repeated values occur at adjacent positions within a list. Compression is achieved by replacing groups of repeated values with one copy of the value, followed by the number of times that it should be repeated. For example, the list `["A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "B", "B", "B", "B", "A", "A", "A", "A", "A", "A", "B"]` would be compressed as `["A", 12, "B", 4, "A", 6, "B", 1]`. Decompression is performed by replicating each value in the list the number of times indicated.

Write a recursive function that decompresses a run-length encoded list. Your recursive function will take a run-length compressed list as its only argument. It will return the decompressed list as its only result. Create a main program that displays a run-length encoded list and the result of decoding it.

Exercise 186: Run-Length Encoding

(*Solved, 38 Lines*)

Write a recursive function that implements the run-length compression technique described in Exercise 185. Your function will take a list or a string as its only argument. It should return the run-length compressed list as its only result. Include a main program that reads a string from the user, compresses it, and displays the run-length encoded result.

Hint: You may want to include a loop inside the body of your recursive function.