ENTRE for EDUCATION in MATHEMATICS and COMPUTING

16: Recursion

We have seen that functions allow us to organize and re-use parts of our code. We have also seen that functions can be defined in terms of other functions. In this lesson we learn that a function can be defined in terms of itself! This very useful approach is called *recursion*. Legend has it that "to understand recursion, you must first understand recursion."

Example

In our lesson on loops, we used a while loop to create the following output.

5

4

3

2

1

Blastoff!

Here is a program that uses recursion to achieve the same effect.

Example

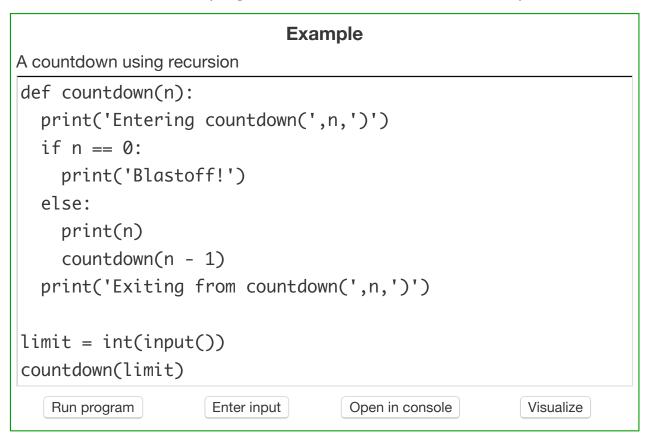
A countdown using recursion

```
def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)

countdown(5)

Run program
Open in console
Visualize
```

Let's add some extra print statements to help us understand how the program works. This version of the program also reads the time limit from input.

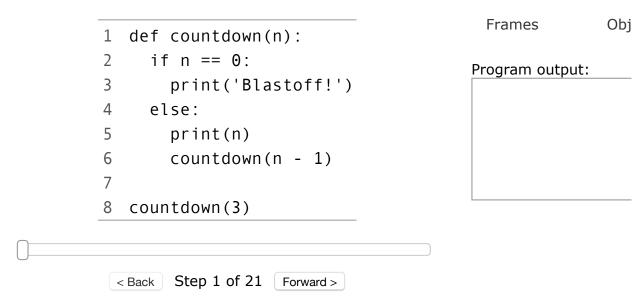


If you like, use **Enter input** with the above program to try other input values. Try 0 first and see what happens, and then 1.

When the input is 5, the program first calls a copy of the countdown function with n=5, which prints 5 and calls countdown(4). This continues until countdown(0), which prints "Blastoff!" and does not call countdown any more. When Python finishes executing the n=0 call of the countdown function, Python returned to the function that called it, which is the n=1 call of

the countdown. Then we return to the n=2 call, and so on.

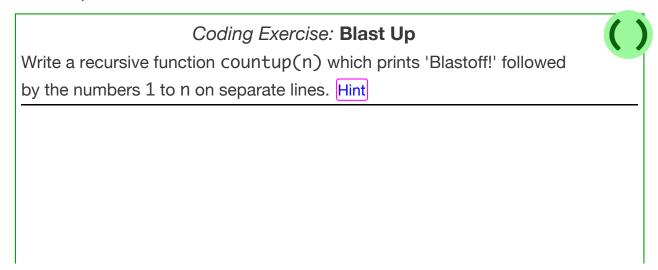
To double-check our understanding, we can also visualize the recursive code:



line that has just executed next line to execute

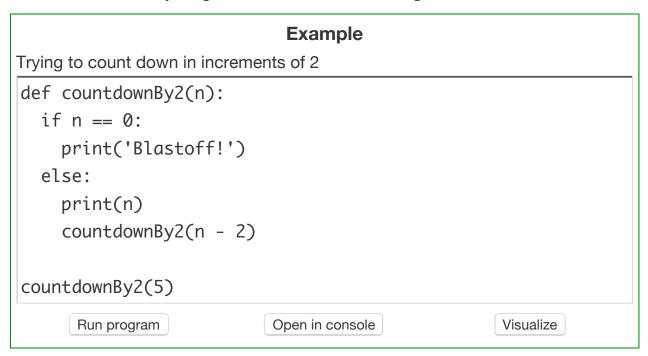
The new twist, which makes recursion unique from the functions we've seen before, is that multiple versions of the function are running at the same time. That is to say, there is more than one frame at a time corresponding to the same function. This is pretty much the same as what we saw in the visualization where one function called another, except now the calling function is the same as the function being called. However, you have to be careful to note that at each step, only the "current" variables (the newest/bottom-most frame) are really used — the non-bottom frames are "paused" and their variables inaccessible.

Now it's your turn to write some code. Modify the countdown function so that it counts up instead of down.



```
1 def countdown(n):
2
     if n == 0:
3
       print('Blastoff!')
4
     else:
5
       print(n)
        countdown(n - 1)
6
7
   Run program
                                       Open in console
                                                             Visualize
                  Enter test statements
More actions...
```

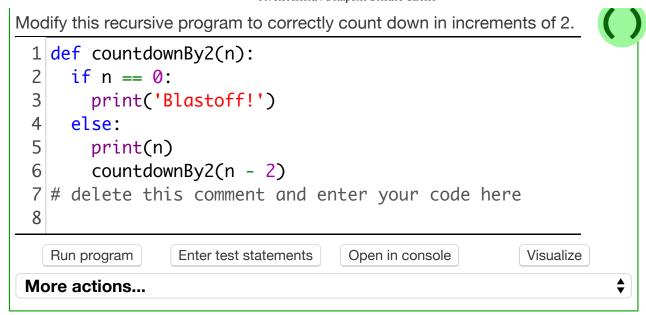
Next, let's modify our countdown program to count in increments of 2. The output should be 5, 3, 1, Blastoff! We will change the function argument from n-1 to n-2. Is there anything else that we need to change?



You can see that this program did not work as we intended. It printed 5, 3, 1, like we wanted, but instead of stopping it continued with -1, -3, -5 and ran forever. (More precisely, it runs out of time and memory, because each recursive call takes up a little more working memory; see the same example in the visualizer.)

When designing a recursive function, we must be careful that its sequence of calls does not continue forever! Modify the countdownBy2 program above so that it correctly stops at 1 (or 2, if n is even) and prints 'Blastoff!'.

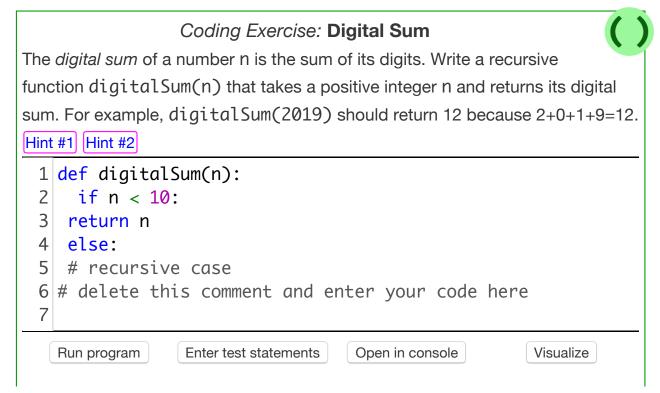




Designing recursive functions

A *recursive function* just means a function that calls itself. But there must be some occasions when the function does not call itself, or else the program will run forever, like we saw above. A **base case** is the part of a recursive function where it doesn't call itself. In the example above, the base case was n<=0. Designing a recursive function requires that you carefully choose a base case and make sure that every sequence of function calls eventually reaches a base case.

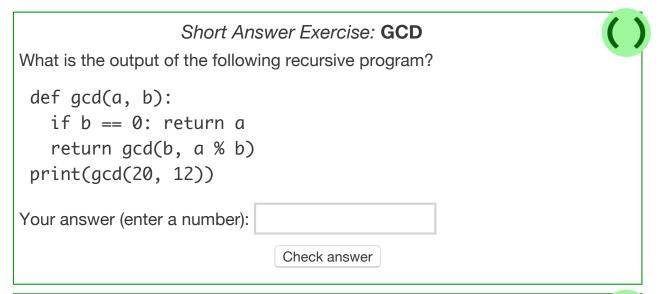
In the next exercise, the base case has been programmed for you, but you will write the rest of the recursive function.



Now you will write a recursive function that calls digitalSum as a subroutine.

Coding Exercise: Digital Root The digital root of a non-negative integer n is computed as follows. Begin by summing the digits of n. The digits of the resulting number are then summed, and this process is continued until a single-digit number is obtained. For example, the digital root of 2019 is 3 because 2+0+1+9=12 and 1+2=3. Write a recursive function digitalRoot(n) which returns the digital root of n. Assume that a working definition of digitalSum will be provided for your program. 1 # delete this comment and enter your code here 2 Run program Enter test statements Open in console Visualize

Exercises



Coding Exercise: Hailstone

The *hailstone* sequence starting at a positive integer n is generated by following two simple rules. If n is even, the next number in the sequence is n/2. If n is odd, the next number in the sequence is 3*n+1. Repeating this process, we generate the hailstone sequence. Write a recursive function hailstone(n) which prints the hailstone sequence beginning at n.

```
Stop when the sequence reaches the number 1 (since otherwise, we would loop
forever 1, 4, 2, 1, 4, 2, ...)
For example, when n=5, your program should output the following sequence:
 5
 16
 8
 4
 2
 1
  1 | # delete this comment and enter your code here
  2
                                          Open in console
                                                                 Visualize
    Run program
                    Enter test statements
                                                                             $
 More actions...
```

Mathematicians believe that every hailstone sequence reaches 1 eventually, no matter what value of n we start with. However, no one has been able to prove this yet.

Nested Lists

Here is an interesting natural application of recursion. A *nested list* is one where you put some lists inside of others, possibly multiple times. For example, some nested lists of integers are [[1, 2], [9, 10]] as well as [[1], 2] and x = [[1], 2, [3, [[4]]]]. The last nested list example is a list with three elements: x[0]==[1] to begin, then x[1]==2, then x[2]==[3, [[4]]]. (So x, viewed as a list, has length 3). Note that a list like [1, 2, 3] also counts as a nested list. **Can we write a function to find the total sum of** *any* **nested list of integers?** For example, on input [[5], 2, [10, 8, [[7]]]] it should return the value 32.

This task is difficult for a while loop or a for loop, since we want a function that works on nested lists with any shape/format. However, nested lists have a naturally recursive structure: a nested list is a list each of whose items is either (a) an integer or (b) a nested list. And, once we compute the sum of each subpart of the main list, the total of those values is the overall sum. We can express this with the following code; it uses isinstance(x, int) which gives a boolean value

telling us whether x has integer type (as opposed to a list).

```
Example: Summing a Nested List
Computing the sum of the elements in a nested list with a recursive function.
Once you press Run you will see its value on some tests.
def nestedListSum(NL):
    if isinstance(NL, int): # case (a): NL is an integ
         return NL
                                    # base case
                                    # case (b): NL is a list o
     sum = 0
    for i in range(0, len(NL)): # add subsums from each pa
         sum = sum + nestedListSum(NL[i])
                                     # all dana
     notiinn ciim
                                     Open in console
   Run program
                     Enter input
                                                        Visualize
```

Recursion is used to break down each nested list into smaller parts? For example, nestedListSum([1, [3, 4], 5]) makes a total of 6 recursive calls: the initial one, then on 1, then on [3, 4], then on 3, then 4, (after which the [3, 4] total is returned as 7) and finally on 5 (after which the overall total 13 is obtained). Here is the same code in the visualizer.

```
Obj
                                                 Frames
 1
   def nestedListSum(NL):
 2
      if isinstance(NL, int):
 3
        return NL
 4
 5
      sum = 0
 6
      for i in range(0, len(NL)):
 7
        sum = sum + nestedListSum(NL[i])
 8
      return sum
 9
10
   # some examples
11
   nestedListSum(129)
   nestedListSum([400, 50, 6])
12
   nestedListSum([[1, 2], [3, 4], 5])
13
14 nestedListSum([[1, [2, 3], 4, 5]])
        < Back | Step 1 of 137 | Forward >
```

line that has just executed next line to execute

Coding Exercise: Searching a Nested List



By writing something similar to nestedListSum, define a recursive function

nestedListContains(NL, target)

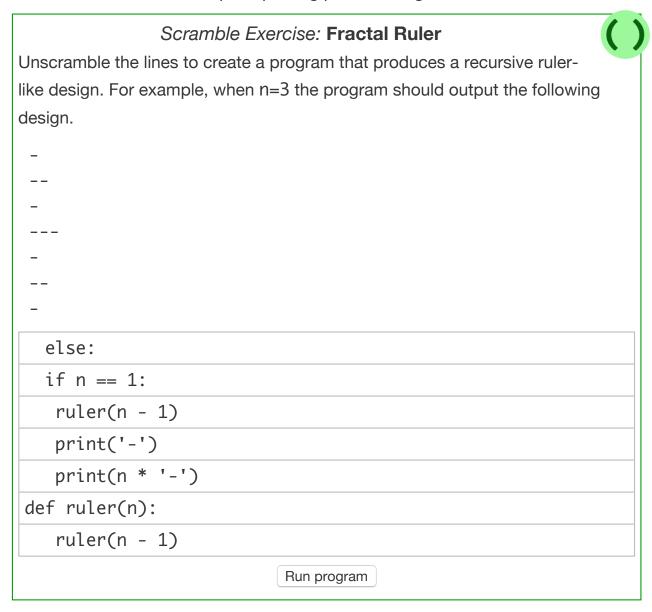
that takes a nested list NL of integers and an integer target, and indicates whether target is contained anywhere in the nested list. Your code should return the boolean value True when it is contained in the nested list, and False if it is not contained in it.

For example, nestedListContains([1, [2, [3], 4]], 3) should give True and nestedListContains([1, [2, [3], 4]], 5) should give False.

| Run program Enter test statements Open in console Visualiz | |
|--|---------------|
| | $\overline{}$ |
| 1 # delete this comment and enter your code here 2 | |
| | |

This Rules

Recursion is also related to fractals — images which contain multiple smaller copies of themselves. The banner at the top of this webpage is one example. The next exercise creates a simple repeating pattern using recursion.



Congratulations! You are ready to move to the next lesson.