Eric Roberts
CSCI 121

Handout #30
October 5, 2018

Introduction to Recursion

This handout offers an introduction to the idea of recursion, which is a fundamental idea in computer science. Most of this material will be in the book somewhere, but my current outline has it scattered through several chapters, including some that I'd already "finished." I thought the best thing to do was put this material on a handout, which you should read for Monday's class:

A simple example of recursion

To gain a better sense of what recursion is, let's imagine that you have been appointed as the funding coordinator for a national campaign organization in a critical election year. Your campaign is long on volunteers and short on cash. Your job is to raise $1,000,000 in contributions so the organization can meet its expenses.

If you know someone who is willing to write a check for the entire $1,000,000, your job is easy. On the other hand, you may not be lucky enough to have friends who are generous millionaires. In that case, you must raise the $1,000,000 in smaller amounts. If the average contribution is $100, you might choose a different tack: call 10,000 friends and ask each of them for $100. But then again, you probably don't have 10,000 friends. So what can you do?

As is often the case when you are faced with a task that exceeds your own capacity, the answer lies in delegating part of the work to others. Your organization has a ready supply of volunteers. If you could find 10 dedicated supporters in different parts of the country and appoint them as regional coordinators, each of those 10 people could then take responsibility for raising $100,000.

Raising $100,000 is simpler than raising $1,000,000, but it hardly qualifies as easy. What should your regional coordinators do? If they adopt the same strategy, they will in turn delegate parts of the job. If they each recruit 10 fundraising volunteers, those people will only have to raise $10,000 each. The delegation process can continue until the volunteers are able to raise the money on their own; because the average contribution is $100, the volunteer fundraisers can probably raise $100 from a single donor, which eliminates the need for further delegation.

If you express this fundraising strategy in pseudocode, it has the following structure:

```
def collectContributions(n):
    if n <= 100:
        Collect the money from a single donor.
    else:
        Find 10 volunteers.
        Get each volunteer to collect n/10 dollars.
        Combine the money raised by the volunteers.
```

The most important thing to notice about this pseudocode translation is that the line

```
Get each volunteer to collect n/10 dollars.
```

is simply the original problem reproduced at a smaller scale. The basic character of the task—raise *n* dollars—remains exactly the same; the only difference is that *n* has a smaller value. Moreover, because the problem is the same, you can solve it by calling the original function. Thus, the preceding line of pseudocode would eventually be replaced with the following line:

```
collectContributions(n / 10)
```

It's important to note that the `collectContributions` function ends up calling itself if the contribution level is greater than $100. In the context of programming, having a function call itself is the defining characteristic of recursion.

The structure of the `collectContributions` function is typical of recursive functions. In general, the body of a recursive function has the following form:

```
if test for simple case:
    Compute a simple solution without using recursion.
else:
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the subproblem solutions into a solution for the whole.
```

This structure provides a template for writing recursive functions and is therefore called the ***recursive paradigm.*** You can apply this technique to programming problems as long as they meet the following conditions:

1. You must be able to identify ***simple cases*** for which the answer is easily determined.
2. You must be able to identify a ***recursive decomposition*** that allows you to break any complex instance of the problem into simpler problems of the same form.

The `collectContributions` example illustrates the power of recursion. As with any recursive technique, the original problem is solved by breaking it down into smaller subproblems that differ from the original only in their scale. Here, the original problem is to raise $1,000,000. At the first level of decomposition, each subproblem is to raise $100,000. These problems are then subdivided in turn to create smaller problems until the problems are simple enough to be solved immediately without recourse to further subdivision. Because the solution depends on dividing hard problems into simpler instances of the same problem, recursive solutions of this form are often called ***divide-and-conquer*** algorithms.

The factorial function

Although the `collectContributions` example illustrates the idea of recursion, it gives little insight into how recursion is used in practice, mostly because the steps that make up the solution, such as finding 10 volunteers and collecting money, are not easily represented in a Python program. To get a practical sense of the nature of recursion, you need to consider problems that fit more easily into the programming domain.

For most people, the best way to understand recursion is to start with simple mathematical functions in which the recursive structure follows directly from the statement of the problem and is therefore easy to see. Of these, the most common is the factorial function (traditionally denoted in mathematics as $n!$), which is defined as the product of the integers between 1 and n. In Python, the equivalent problem is to write an implementation of a function with the prototype

```
def fact(n)
```

that takes an integer **n** and returns its factorial.

As you know from the earlier chapters, it is easy to code the **fact** function using a **for** loop, as illustrated by the following implementation:

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

This implementation uses a **for** loop to cycle through each of the integers between 1 and **n**. In the recursive implementation, this loop does not exist. The same effect is generated instead by the cascading recursive calls.

Implementations that use looping (typically by using **for** and **while** statements) are said to be ***iterative.*** Iterative and recursive strategies are often seen as opposites because they can be used to solve the same problem in rather different ways. These strategies, however, are not mutually exclusive. Recursive functions sometimes employ iteration internally.

The recursive formulation of `fact`

The iterative implementation of **fact**, however, fails to take advantage of an important mathematical property of factorials. Each factorial is related to the factorial of the next smaller integer in the following way:

$$n! = n \times (n - 1)!$$

Thus, $4!$ is $4 \times 3!$, $3!$ is $3 \times 2!$, and so on. To make sure that this process stops at some point, mathematicians define $0!$ to be 1. Thus, the conventional mathematical definition of the factorial function looks like this:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

This definition is recursive, because it defines the factorial of n in terms of the factorial of $n - 1$. The new problem—finding the factorial of $n - 1$—has the same form as the original problem, which is the fundamental characteristic of recursion. You can then use the same process to define $(n - 1)!$ in terms of $(n - 2)!$. Moreover, you can carry this process forward step by step until the solution is expressed in terms of 0!, which is equal to 1 by definition.

From your perspective as a programmer, the practical impact of the mathematical definition is that it provides a template for a recursive implementation. In Python, you can implement a function **`fact`** that computes the factorial of its argument as follows:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

If **`n`** is 0, the result of **`fact`** is 1. If not, the implementation computes the result by calling **`fact(n - 1)`** and then multiplying the result by **`n`**. This implementation follows directly from the mathematical definition of the factorial function and has precisely the same recursive structure.

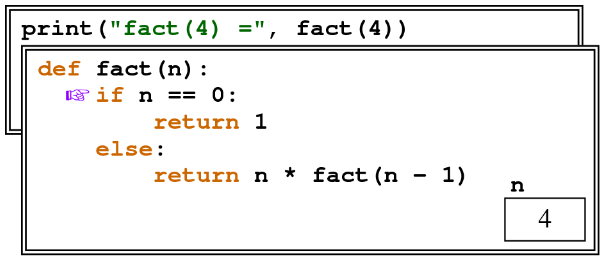
Tracing the recursive process

If you work from the mathematical definition, writing the recursive implementation of **`fact`** is straightforward. On the other hand, even though the definition is easy to write, the brevity of the solution may seem suspicious. When you are learning about recursion for the first time, the recursive implementation of **`fact`** seems to leave something out. Even though it clearly reflects the mathematical definition, the recursive formulation makes it hard to identify where the actual computational steps occur. When you call **`fact`**, for example, you want the computer to give you the answer. In the recursive implementation, all you see is a formula that transforms one call to **`fact`** into another one. Because the steps in that calculation are not explicit, it seems somewhat magical when the computer gets the right answer.

If you trace through the logic the computer uses to evaluate any function call, however, you discover that no magic is involved. When the computer evaluates a call to the recursive **`fact`** function, it goes through the same process it uses to evaluate any other function call. To visualize the process, suppose that you have executed the statement

```
print("fact(4) =", fact(4))
```

in the IDLE interpreter. When this statement calls **`fact`**, Python creates a new stack frame and copies the argument value into the formal parameter **`n`**. The frame for **`fact`** temporarily supersedes the frame executing the **`print`** call as shown in the following diagram:

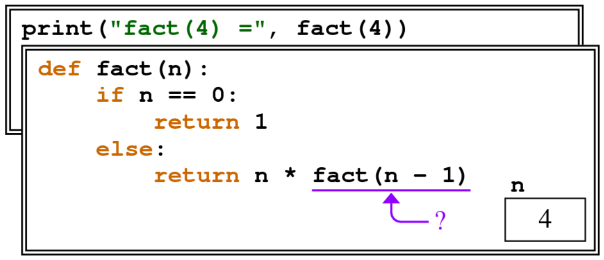


In the diagram, the code for the body of **fact** is shown inside the frame to make it easier to keep track of the current position in the program.

The computer now begins to evaluate the body of the function, starting with the **if** statement. Because **n** is not equal to 0, control proceeds to the **else** clause, where the program must evaluate and return the value of the expression

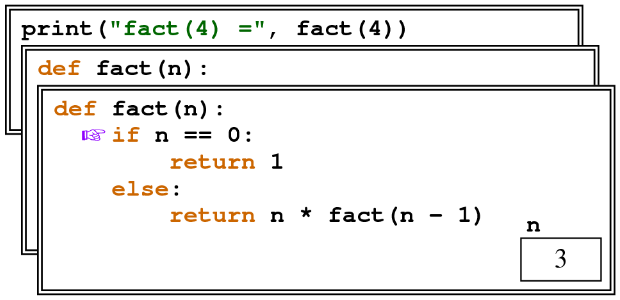
```
n * fact(n - 1)
```

Evaluating this expression requires computing the value of **fact(n - 1)**, which introduces a recursive call. When that call returns, all the program has to do is to multiply the result by **n**. The current state of the computation can therefore be diagrammed as follows:



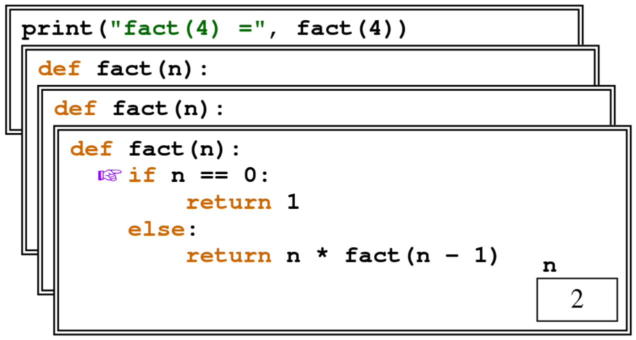
As soon as the call to **fact(n - 1)** returns, the result is substituted for the expression underlined in the diagram, which allows computation to proceed.

The next step in the computation is to evaluate the call to **fact(n - 1)**, beginning with the argument expression. Because the current value of **n** is 4, the argument expression **n - 1** has the value 3. The computer then creates a new frame for **fact** in which the formal parameter is initialized to this value. Thus, the next frame looks like this:



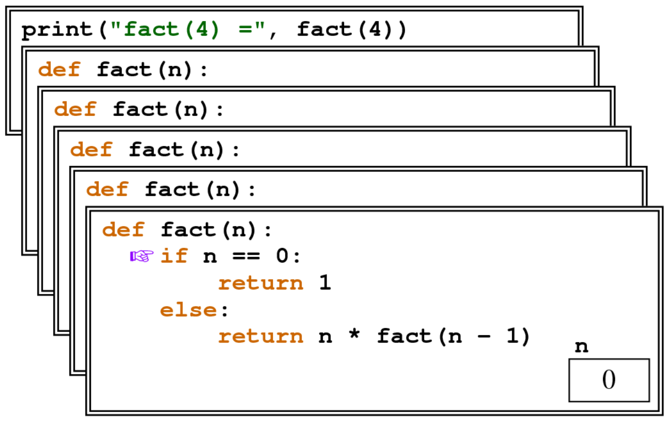
There are now two frames labeled **`fact`**. In the most recent one, the computer is just starting to calculate **`fact(3)`**. This new frame hides the previous frame for **`fact(4)`**, which will not reappear until the **`fact(3)`** computation is complete.

Computing **`fact(3)`** again begins by testing the value of **`n`**. Since **`n`** is still not 0, the **`else`** clause instructs the computer to evaluate **`fact(n - 1)`**. As before, this process requires the creation of a new stack frame, as shown:



```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
def fact(n):
  ☞if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Following the same logic, the program must now call **`fact(1)`**, which in turn calls **`fact(0)`**, creating two new stack frames, as follows:

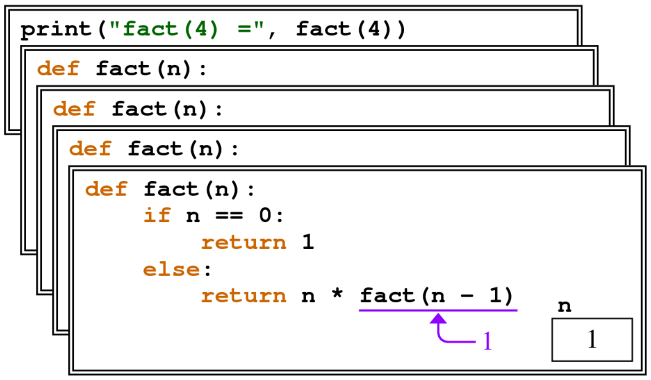


```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
def fact(n):
def fact(n):
def fact(n):
  ☞if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

At this point, however, the situation changes. Because the value of **`n`** is 0, the function can return its result immediately by executing the statement

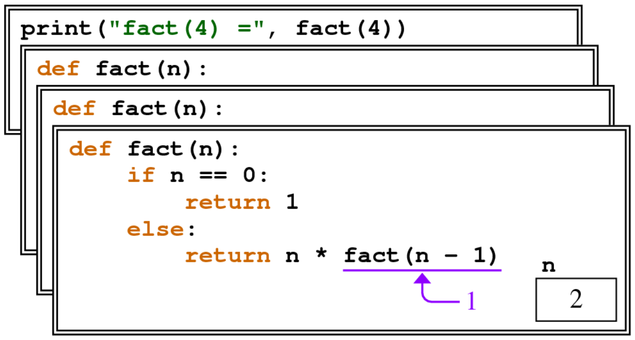
```
return 1;
```

The value 1 is returned to the calling frame, which resumes its position on top of the stack, as shown:



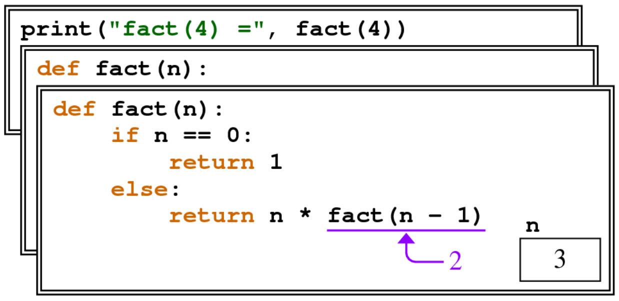
```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
def fact(n):
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

From this point, the computation proceeds back through each of the recursive calls, completing the calculation of the return value at each level. In this frame, for example, the call to **fact(n - 1)** can be replaced by the value 1, as shown in the diagram for the stack frame. In this stack frame, **n** has the value 1, so the result of this call is simply 1. This result gets propagated back to its caller, which is represented by the top frame in the following diagram:



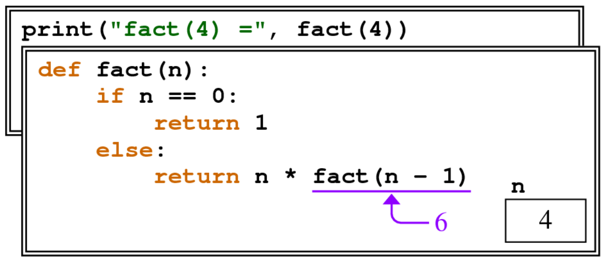
```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Because **n** is now 2, evaluating the **return** statement causes the value 2 to be passed back to the previous level, as follows:



```
print("fact(4) =", fact(4))
def fact(n):
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

At this stage, the program returns 3×2 to the previous level, so that the frame for the initial call to **fact** looks like this:



```
print("fact(4) =", fact(4))
```

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

The final step in the calculation process consists of calculating 4 × 6 and returning the value 24 to the main program.

The recursive leap of faith

The point of including the complete trace of the **`fact(4)`** computation is to convince you that the computer treats recursive functions just like all other functions. When you are faced with a recursive function, you can—at least in theory—mimic the operation of the computer and figure out what it will do. By drawing all the frames and keeping track of all the variables, you can duplicate the entire operation and come up with the answer. If you do so, however, you will usually find that the complexity of the process ends up making the computation much harder to follow.

Whenever you try to understand a recursive program, it is useful to put the underlying details aside and focus instead on a single level of the operation. At that level, you are allowed to assume that any recursive call automatically gets the right answer as long as the arguments to that call are in some sense simpler than the original arguments. This psychological strategy—assuming that any simpler recursive call will work correctly—is called the ***recursive leap of faith.*** Learning to apply this strategy is essential to using recursion in practical applications.

As an example, consider what happens when this implementation is used to compute **`fact(n)`** with **`n`** equal to 4. To do so, the recursive implementation must compute the value of the expression

```
n * fact(n - 1)
```

By substituting the current value of **`n`** into the expression, you know that the result is

```
4 * fact(3)
```

Stop right there. Computing **`fact(3)`** is simpler than computing **`fact(4)`**. Because it is simpler, the recursive leap of faith allows you to assume that it works. Thus, you should assume that the call to **`fact(3)`** will correctly compute the value of 3!, which is 3 × 2 × 1, or 6. The result of calling **`fact(4)`** is therefore 4 × 6, or 24.

As you look at the examples in the rest of this chapter, try to focus on the big picture instead of the details. Once you have made the recursive decomposition and identified the simple cases, be satisfied that the computer can handle the rest.