**Introduction to Recursion**

Let's explore a bit of recursion as one of our last topics of the course! Recursion will be covered in further courses, but let's get our feet wet for now.

**What is Recursion?**

In programming, [Recursion](https://en.wikipedia.org/wiki/Recursion) is when a **method calls itself**. As we have seen during the course, using helper methods can be very useful when it comes to solving problems. In our classic helper method examples, we have commonly called methods from other methods. Recursion is a similar process except we are calling the **same method**.

As we explore recursion it is important to keep in mind all of the previous concepts we learned about methods. All those method concepts will still apply to recursion!

**A Simple Recursion**

Let's write our very first (albeit broken) recursive method:

# recursive method definition:

def say\_hello

p "hello"

say\_hello

end

say\_hello # prints "hello" forever

Notice that the say\_hello method definition contains a call to itself. A recursive method definition still obeys all the evaluation rules as a normal method, so like usual, the code will not run until we *call* it.

Let's step through how this code evaluates. For clarity, we'll be numbering the calls to say\_hello. The initial call will be number 0:

* When we call say\_hello for the first time (say\_hello\_0), we run the code inside the definition as usual. That means we print out "hello" and call say\_hello again (say\_hello\_1). Now we need to evaluate the call to say\_hello\_1.
* say\_hello\_1 prints "hello" and calls say\_hello\_2, now we need to evaluate say\_hello\_2
* say\_hello\_2 prints "hello" and calls say\_hello\_3
* say\_hello\_3 prints "hello" and calls say\_hello\_4
* ... and this pattern continues forever!

Our say\_hello method enters an infinite loop where one call to the method triggers another call. And that call triggers yet another call, etc.. If you run this code, you will enter an infinite loop. However, the program will crash with a SystemStackError: stack level too deep. Whenever we call a method, some of our system's memory must be allocated to execute that method call. This is known as adding to the stack. Since our say\_hello code continuously calls methods forever, we will run out of space on the stack (run out of memory) and crash!

**Recursive Countdown**

In our previous example we saw how our recursive method crashed because it entered an infinite recursive loop. Of course useful recursive methods should not crash, so let's go through the process of building a working one.

Let's build a recursive countDown that starts ticking down numbers:

def count\_down(num)

p num;

count\_down(num - 1)

end

count\_down(10) # this prints decreasing numbers starting at 10 forever

This recursive definition evaluates in a similar way to before, however, now we are passing in decreasing numbers. For any num, every call to countDown(num) will call countDown(num - 1), starting from our initial call to countDown(10):

* countDown(10) prints 10 and calls countDown(9), so next we evaluate countDown(9)...
* countDown(9) prints 9 and calls countDown(8)...
* countDown(8) prints 8 and calls countDown(7)...
* ... and this process continues forever!

In your mind imagine these successive calls:

countDown(10) -> countDown(9) -> countDown(8) ...

Our countdown crashes with a similar error as last time because we entered an infinite loop again. Hmmm, what if we modify our method so that it stops at 0:

def count\_down(num)

if num == 0

p "Houston, we have lift-off!"

return;

end

p num

count\_down(num - 1)

end

count\_down(10) # prints numbers from 10 to 1, and finally "Houston, we have lift-off!"

Now our method stops once we hit 0! Let's say we get to the point where we evaluate countDown(0). That means that the if condition is true, so we print the lift-off message and return. Recall that as soon as we hit a return we exit that method call immediately. Since we immediately return out of our call to countDown(0), countDown(0) never calls countDown(-1), breaking our recursive loop!

**Anatomy of a Recursive Method**

In recursive methods, we need to implement a way to stop the recursive loop and prevent it from looping forever. We took care of the infinite loop issue in our countDown by using an if statement that prevents another recursive call. In general, we call such a statement the **base case**

A recursive method consists of two fundamental parts:

* the **base case** where we halt the recursion by not making a further call
* the **recursive step** where we continue the recursion by making another subsequent call

def count\_down(num)

# base case

if num == 0

p "Houston, we have lift-off!"

return;

end

p num

# recursive step

count\_down(num - 1)

end

Next up, let's solve some recursive problems!

**factorial**

Recall our factorial problem from earlier in the course:

# Write a method `factorial(n)` which takes a number and returns the factorial of n.

# A factorial is the product of all whole numbers between 1 and n, inclusive.

# For example, `factorial(5)` is 5 \* 4 \* 3 \* 2 \* 1 = 120.

How can we solve this problem using recursion? Notice that the structure of factorial has us take decreasing numbers similar to the countDown. However, this time we need to keep multiplying them together.

If we lay out the math used to calculate the factorial of some numbers, we'll notice a pattern:

# factorial(5) = 5 \* 4 \* 3 \* 2 \* 1

# factorial(4) = 4 \* 3 \* 2 \* 1

# factorial(3) = 3 \* 2 \* 1

# factorial(2) = 2 \* 1

# factorial(1) = 1 (base case)

In the outline above, notice that as the input number becomes smaller and smaller, the problem we solve also becomes smaller. There are less multiplications that need to take place!

Let's see the pattern programmatically. We can find the factorial of a number by using the factorial of another number:

# factorial(5) = 5 \* factorial(4)

# factorial(4) = 4 \* factorial(3)

# factorial(3) = 3 \* factorial(2)

# factorial(2) = 2 \* factorial(1)

# factorial(1) = 1 (base case)

Or in general, if n is some number:

* factorial(n) = n \* factorial(n - 1)
* factorial(1) = 1

Now let's implement factorial with some recursive Ruby:

def factorial(n)

return 1 if n == 1

n \* factorial(n - 1);

end

factorial(5) # => 120

Note that mathematically, factorial(0) = 1 so we could have also used that as the base case.

**Solving a Problem Recursively:**

Because every recursive problem must have a base and recursive case, we can follow these steps to help us write a recursive method:

1. Identify the base case in the problem and code it. The base case should explicitly handle the scenario(s) where the arguments are so trivially "small", that we immediately know the result without further calculation. Be sure it works by testing it.
2. Solve the next level of the problem, using the result of the base case. Test it.
3. Modify the code in step 2, generalizing it for every level of the problem

### fib

Let's take a look at a classic recursive problem. The fibonacci sequence!

# Write a method fib(n) that takes in a number and returns the nth number of

# the fibonacci sequence.

# In the fibonacci sequence, the 1st number is 1 and the 2nd number is 1. To generate the

# next number in the sequence, we take the sum of the previous two fibonacci numbers.

# For example the first 5 numbers of the fibonacci sequence are `1, 1, 2, 3, 5`

# Examples:

# fib(1) # => 1

# fib(2) # => 1

# fib(3) # => 2

# fib(4) # => 3

# fib(5) # => 5

# fib(6) # => 8

# fib(7) # => 13

To get a fibonacci number, we need to take the sum of the previous two. Take a look at the following ways we can describe fib.

# fib(5) = fib(4) + fib(3)

# fib(4) = fib(3) + fib(2)

# fib(3) = fib(2) + fib(1)

# fib(2) = 1 base case

# fib(1) = 1 base case

In general:

fib(n) = fib(n - 1) + fib(n - 2)

Finally, let's implement fib using code:

def fib(n)

return 1 if n == 1 || n == 2

fib(n - 1) + fib(n - 2)

end

This should feel like magic! To make sense of recursive code like fib, use abstraction and your comfort with helper methods. Recursion is only "different" from using regular helper methods because we are using the same method as the helper. However, you can use abstraction in the same way. If we want to solve fib(5) we can decompose it into fib(4) + fib(3)! Take a second to appreciate the beauty of our recursive fib method! So cool.

### When is recursion appropriate?

Recursion allows us to solve problems in an elegant way. However, recursion is a tool that is only appropriate for certain problems. Look to the structure of a problem to figure out if it can be solved recursively. **Recursion is used to solve problems that can be decomposed into smaller versions of the**same**problem.** For example we can decompose fib(n) into fib(n - 1) + fib(n - 2). Intuitively, we know that fib(n - 1) is a "smaller" or "easier" problem than fib(n). The easiest subproblem is fib(1) or fib(2) because the answer is simply 1; this is an assumption in the fibonacci sequence. We use the easiest subproblems as the base case in recursion.