5.6. Stack Frames: Implementing Recursion

Suppose that instead of concatenating the result of the recursive call to `toStr` with the string from `convertString`, we modified our algorithm to push the strings onto a stack instead of making the recursive call. The code for this modified algorithm is shown in ActiveCode 1.

Run Load History

```
from pythonds.basic import Stack

rStack = Stack()

def toStr(n,base):
    convertString = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            rStack.push(convertString[n])
        else:
            rStack.push(convertString[n % base])
        n = n // base
    res = ""
    while not rStack.isEmpty():
        res = res + str(rStack.pop())
```

Activity: 5.6.1 Converting an Integer to a String Using a Stack (lst_recstack)

Each time we make a call to `toStr`, we push a character on the stack. Returning to the previous example we can see that after the fourth call to `toStr` the stack would look like Figure 5. Notice that now we can simply pop the characters off the stack and concatenate them into the final result, `"1010"`.

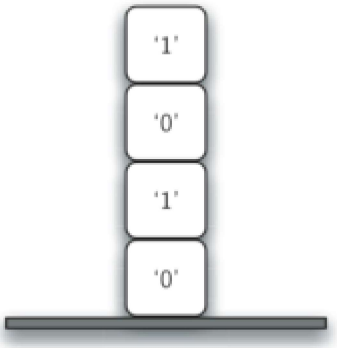

Figure 5: Strings Placed on the Stack During Conversion

The previous example gives us some insight into how Python implements a recursive function call. When a function is called in Python, a **stack frame** is allocated to handle the local variables of the function. When the function returns, the return value is left on top of the stack for the calling function to access. Figure 6 illustrates the call stack after the return statement on line 4.

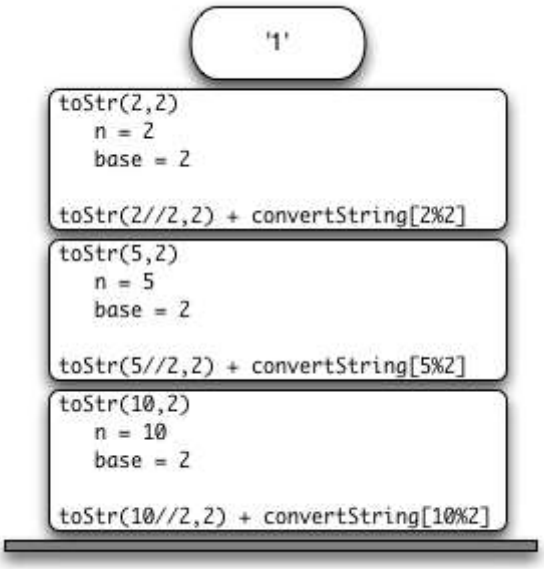

Figure 6: Call Stack Generated from `toStr(10,2)`

Notice that the call to `toStr(2//2,2)` leaves a return value of `"1"` on the stack. This return value is then used in place of the function call (`toStr(1,2)`) in the expression `"1" + convertString[2%2]` , which will leave the string `"10"` on the top of the stack. In this way, the Python call stack takes the place of the stack we used explicitly in Listing 4. In our list summing example, you can think of the return value on the stack taking the place of an accumulator variable.

The stack frames also provide a scope for the variables used by the function. Even though we are calling the same function over and over, each call creates a new scope for the variables that are local to the function.

You have attempted 1 of 2 activities on this page

user not logged in

(pythondsConvertinganIntegertoaStringinAnyBase.html)

(pythondsintro-VisualizingRecurs