CSC108: Visualizing Function Calls

This supplemental document discusses how function calls are managed in computer memory. Specifically, it deals with the concepts of call stack and stack frames, concepts which can be complicated to grasp at first.

Similar information can be found in the lecture summary of the "Visualizing Function Calls" video lecture on Coursera: https://d396qusza40orc.cloudfront.net/programming1%2Flecture_summaries%2Fweek2%2Ffuncvis.html

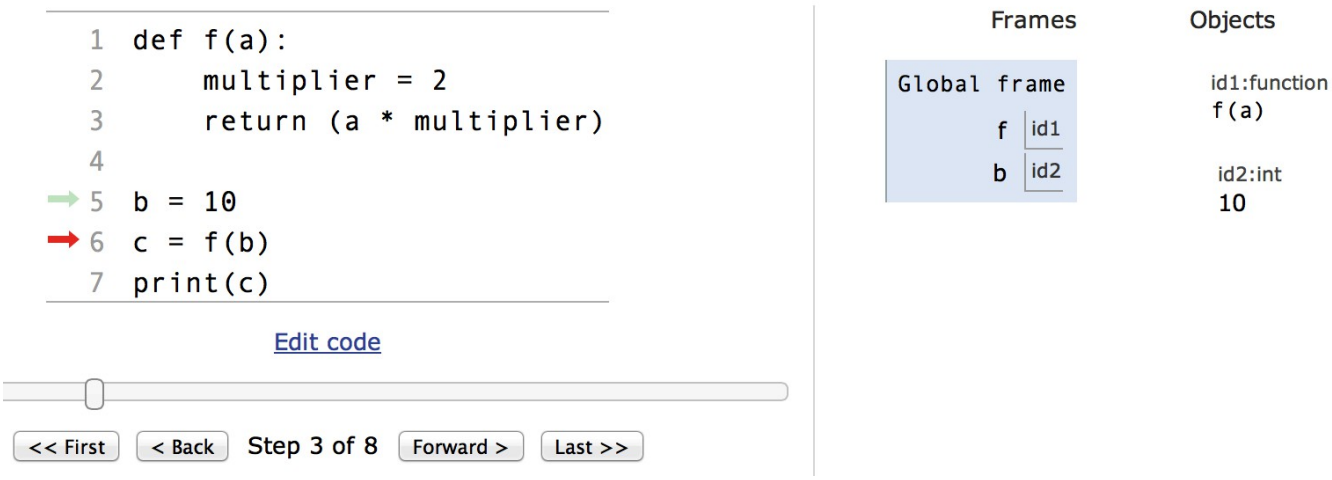
Example 1: Visualizing a single function call

Let's assume we have the following Python program, which you can visualize here: http://tinyurl.com/lpvuz78

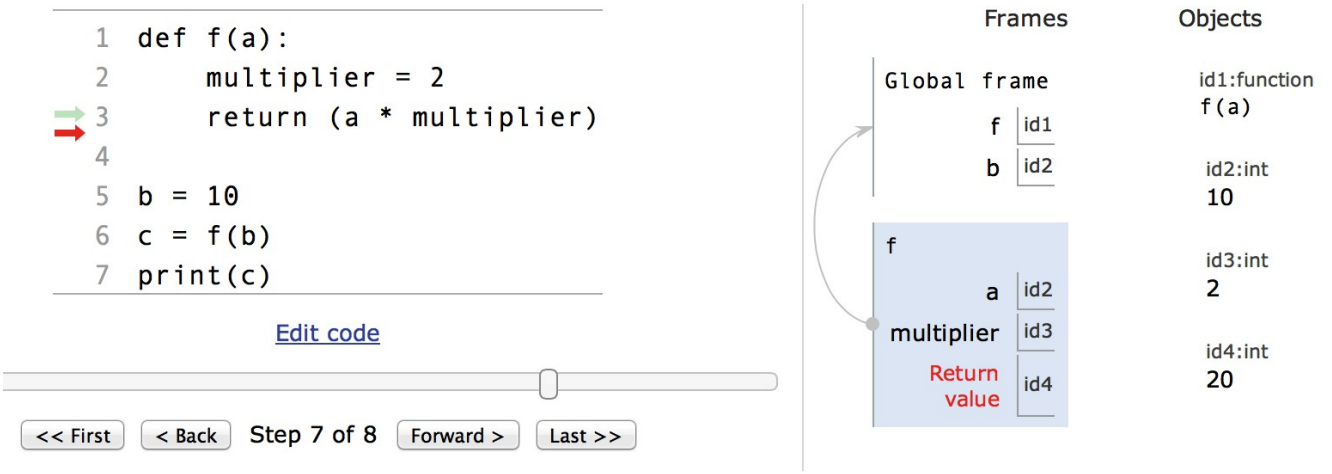
```
def f(a):
    multiplier = 2
    return (a * multiplier)

b = 10
c = f(b)
print(c)
```

Remember that in the visualizer window there are two distinct areas: one for **frames** and one for **objects.** When Python reads a function definition, e.g., for function f(a) in our case, it creates a variable with the name f in the frames section. This variable contains the memory address of a function object which resides in the objects section, i.e., in memory. The function object contains all necessary information for this function, such as the code from the function's body, the function parameters (if any), the docstring and others. Once the Python has read the function definition, the state of our memory model is as follows:

Notice that there is a global frame (the main frame in your program) inside which variable f is created. Once we execute the assignment statement b = 10, a new variable b will also be created inside this global frame.

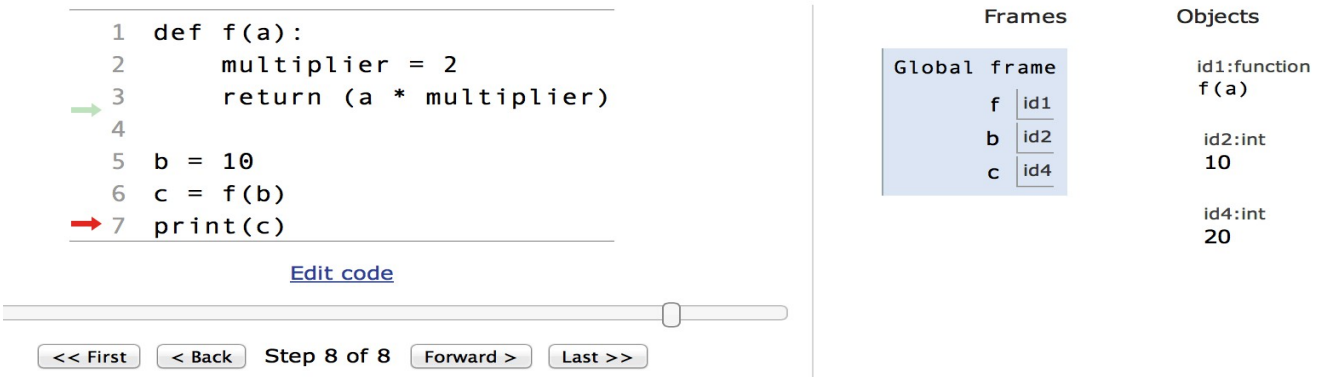


The next Python instruction contains a function call to function f with an argument b. Remember that the right-hand side of the assignment is evaluated first. The moment we call function f, a new stack frame will be created for function f. This will be underneath the existing global frame.
Note that this new frame will contain any variables that are specific to this function such as its parameter *a*, the local variable *multiplier* etc.



The above figure shows all the contents of the stack frame for function f. There is an arrow from this frame to the global frame which denotes that function f was called from the main program (global frame).

Once function f returns, its frame is removed. The variable c, which is created in the global frame, contains the memory address id4 (the one returned by function f).

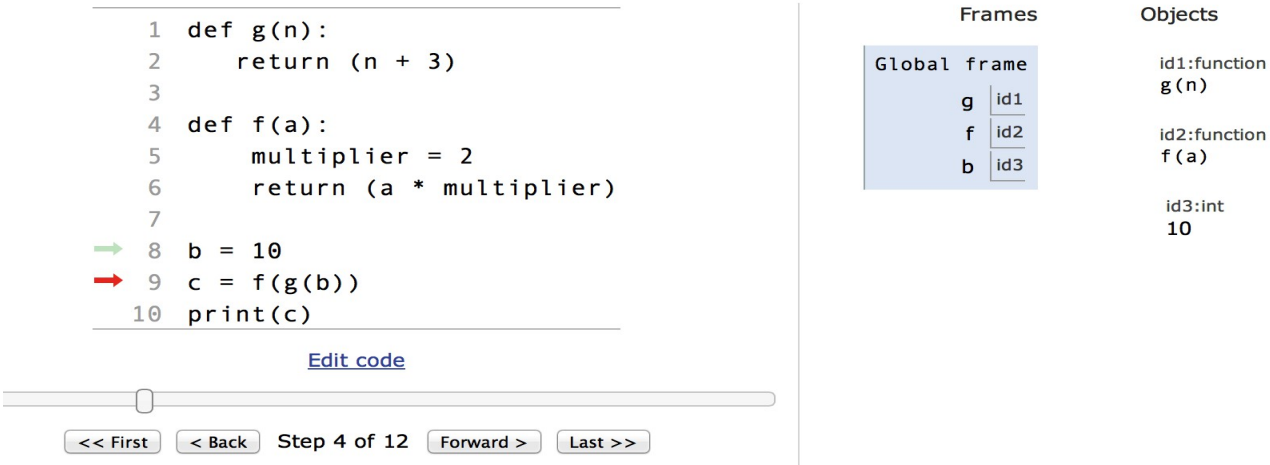


Example 2: Visualizing nested function calls

We will now extend the previous example to include nested function calls. We define a new function g(n) and we modify the function call of f from f(b) to f(g(b)).

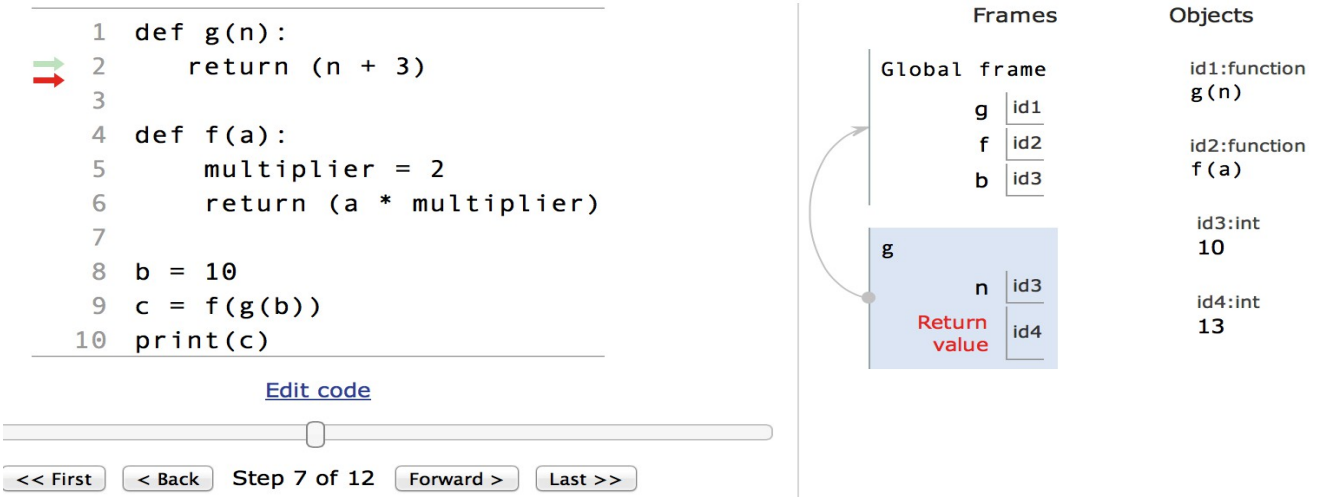
```
def g(n):
  return (n + 3)

def f(a):
   multiplier = 2
   return (a * multiplier)

b = 10
c = f(g(b))
print(c)
```

We fast forward until the statement b = 10 has executed. At this point, the visualizer shows the following:

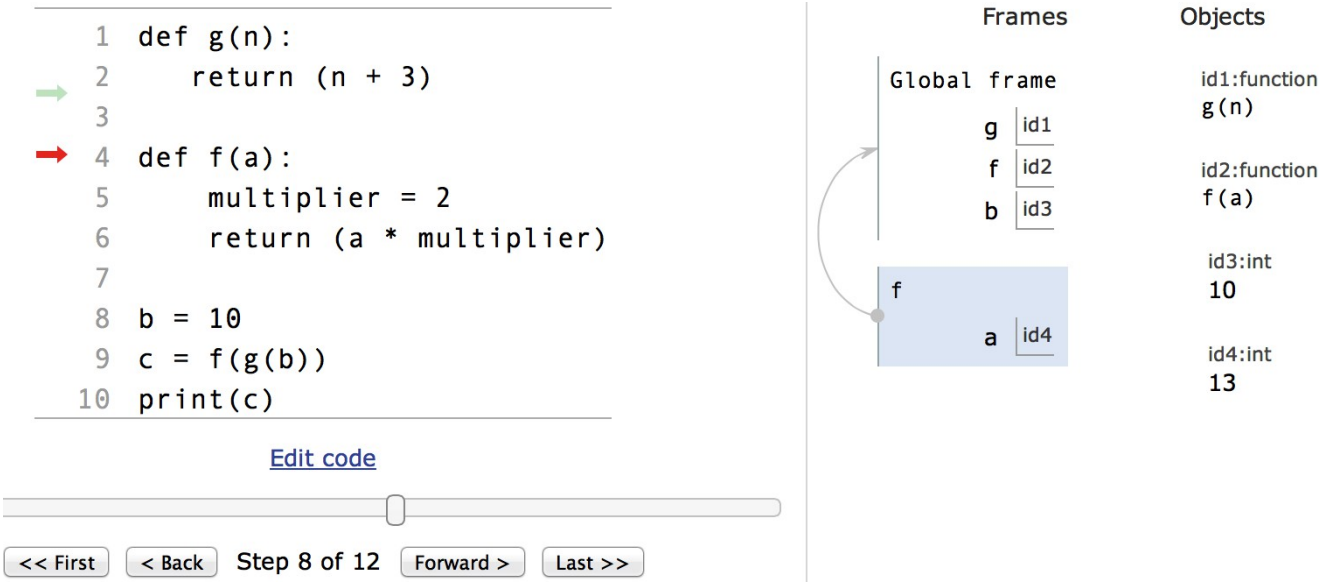


The global frame now has two variables (g and f) which contain the memory addresses of their respective function objects. Variable b which refers to value 10 also appears in the global frame.

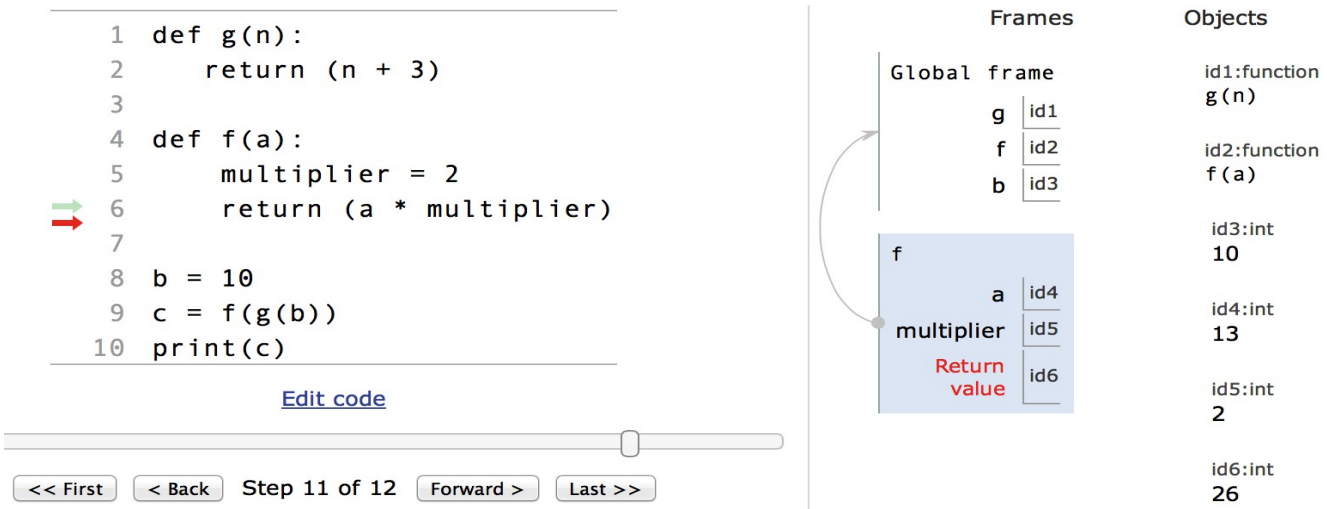
In order to evaluate the right hand side of c = f(g(b)), we first need to evaluate g(b). So the stack frame for function g will appear first (i.e., before the stack frame for function f).



Once function g returns, its stack frame is removed too and the stack frame for function f appears.



Just before function f returns, the visualizer tool shows the following:



Finally, once function f returns, its stack frame is removed too and a new variable c is created in the global frame.