Higher-Order Environments, Currying, and Recursion Solutions

Computer Science Mentors 61A

February 6–February 10, 2023

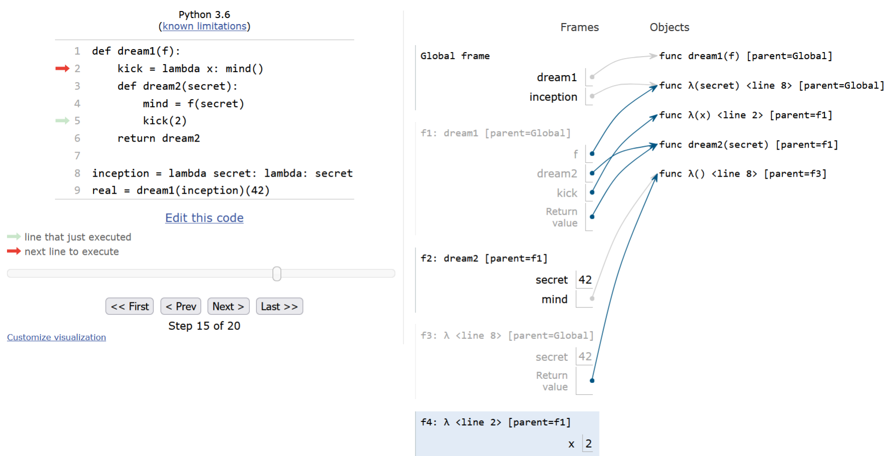
Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

1. Draw the environment diagram that results from running the code.

```
def dream1(f):
    def dream2(secret):
        mind = f(secret)
        kick = lambda x: mind()
        return kick(2)
    return dream2

inception = lambda secret: lambda: secret
real = dream1(inception)(42)
```

Output: 42



https://imgur.com/a/ZKwZzdy

2. Draw the environment diagram that results from running the code.

```
def a(y):
    d = 1
    b = lambda x: y(x)
    e = lambda x: x(3)
    return e(b)
d = 5
a(lambda x: 4 - x + d)
```

https://goo.gl/9vxEwv

3. Implement `compound`, which takes in a single-argument function `base_func` and returns a two-argument compounder function `g`. The function `g` takes in an integer `x` and positive integer `n`.

   Each call to `g` will print the result of calling `f` repeatedly 0,1,.., n-1 times on `x`. That is, `g(x, 2)` prints `x`, then `f(x)`. Then, `g` will return the next two-argument compounder function.

```
def compound(base_func, prev_compound=lambda x: x):
    """
    >>> add_one = lambda x: x + 1
    >>> adder = compound(add_one)
    >>> adder = adder(3, 2)
    3       # 3
    4       # f(3)
    >>> adder = adder(4, 4)
    6       # f(f(4))
    7       # f(f(f(4)))
    8       # f(f(f(f(4))))
    9       # f(f(f(f(f(4)))))
    """
    def g(x, n):
      new_comp = ____________________________
      while n > 0:
        print(______________________________)
        new_comp = (lambda save_comp: \
                        _______________________)(____________)
        ____________________________________________________
      return _______________________________________________
    return ____________________________________
```

```
def compound(base_func, prev_compound=lambda x : x):
  def g(x, n):
    new_comp = prev_compound
    while n > 0:
      print(new_comp(x))
      new_comp = (lambda save_comp: \
                  lambda x: base_func(save_comp(x)))(new_comp)
      n -= 1
    return compound(base_func, new_comp)
  return g
```

There are three steps to writing a recursive function:

1. Create base case(s)
2. Reduce your problem to a smaller subproblem and call your function recursively to solve the smaller subproblem(s)
3. Use the subproblems' solutions as pieces to construct a larger problem's solution (This can happen in many layers!)

Real World Analogy for Recursion

Imagine that you're in line for boba, but the line is really long, so you want to know what position you're in. You decide to ask the person in front of you how many people are in front of them. That way, you can take their response and add 1 to it to find your place. Now, the person in front of you is faced with the same problem that you were trying to solve, with one less person in front of them than you. They decide to take the same approach that you did by asking the person in front of them. This continues until the very first person in line is asked. At this point, the person at the front knows that there are 0 people in front of them, so they can tell the person behind them that there are 0 people in front. Now, the second person can figure out that there is 1 person in front of them, and can relay that back to the person behind them, and so on, until the answer reaches you.

Looking at this example, we see that we have broken down the problem of "how many people are there in front of me?" to 1 + "how many people are there in front of the person in front of me"? This problem will terminate with the person at the front of the line (with 0 people in front of them). Putting this into more formal terms, we are breaking down the problem into a **recurrence relationship**, and the termination case (when the question gets to the very first person in line) is called the **base case**.

In addition to this, we can also imagine a scenario that can help us understand how recursion functions. Imagine you're on a linear hiking trail in which you go back the way you came from. In case you get lost, you place markers down at each point on the trail until you reach the very end, the "base" of the trail. Once you reach the end of the trail, you get curious as to how many markers you have. While coming back, you pick up and count each marker along the way until you reach the start.

Similar to the boba example, we break down the problem of "how many markers" to 1 + the next marker until we reach where we started. What's important to understand from this example is that as a program goes through recursion, it doesn't formally solve the problem until it reaches the base case, in which case it works its way up from the base case to the original input to construct your final answer. Just like you, the program goes out (down the call stack), marker by marker (call by call) and back, solving the problem.

4. What is wrong with the following function? How can we fix it?

```
def factorial(n):
    return n * factorial(n)
```

There is no base case and the recursive call is made on the same n.

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

5. Complete the definition for num_digits, which takes in a number n and returns the number of digits it has.

```
def num_digits(n):
    """Takes in an positive integer and returns the number of
    digits.

    >>> num_digits(0)
    1
    >>> num_digits(1)
    1
    >>> num_digits(7)
    1
    >>> num_digits(1093)
    4
    """


    if n < 10:
        return 1
    else:
        return 1 + num_digits(n // 10)
```