Higher-Order Environments, Currying, and Recursion Meta

Computer Science Mentors 61A

September 16–September 20, 2024

Recommended Timeline

- HOFs mini-lecture/review - 5 mins
- Inception OR ABDE - 10 mins (check in with your students to see how they feel about the general structure of higher order functions; do this if they feel a bit shaky)
- Compound - 15 mins
- General recursion mini-lecture - 10 mins (Remember, recursion can be a challenging concept when encountered for the first time. It's perfectly fine to spend extra time on this topic if needed to ensure students grasp the fundamentals.)
- Wrong factorial - 5 to 10 mins
- num_digits - 5 mins

Please remember, there is no expectation that you get through all problems in a section. Pick the most pertinent problems for your section. Lastly, note that page 5 of the student-facing worksheet has been left blank in case they need extra scratch paper to work on any of the problems.

Created by Jim Fang, Bryan Adolfo Ventura Benitez. Worksheet feedback: links.jimfang.dev/feedback

1 Higher-Order Functions cont.

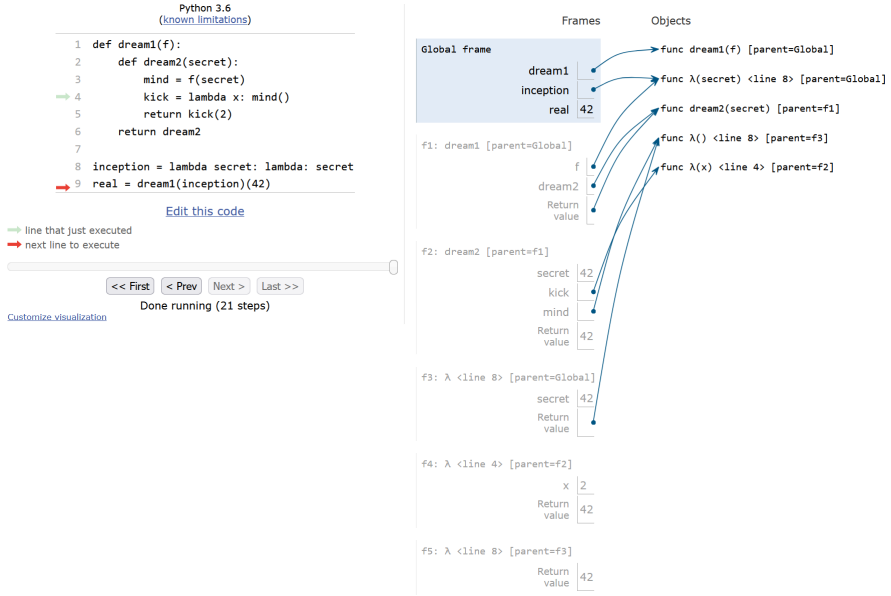
1. Draw the environment diagram that results from running the code below.

   It may be helpful to give students an overview on the rules of drawing environment diagrams, such as when to open new frames, how to format the frames, what parent frames are, when to draw pointers to objects, frame hierarchy, lambdas, and any tips/tricks you may think of!

```
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
```

Inception and ABDE are very similar in terms of the skills they test. If your students are finding this concept challenging, focus on these problems during your section before moving on to the HOF challenge problems. It's important to work on either Compound or Partial Summer, as they both cover essential aspects of higher-order functions. Since these problems are more challenging, consider working on the rest of the worksheet first and then dedicate time to thoroughly working through one of these challenge problems.

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.

Teaching Tips

1. Base Case - What is the simplest case? Or in what case do you want your recursion to stop? It's helpful to use edge cases to nudge students if they get stuck.
2. Break the problem down into smaller problems (Try to address this in terms of each specific problem, then extrapolate for general understanding)
   - What do you need to do to reach your base case?
   - For example: in factorial (usually seen in lecture), we have to subtract by one each time we do a recursive call
3. Solve the smaller problem recursively
   - How would you use the solution to the smaller problem to write a solution to the original problem?
   - "Recursive Leap of Faith"—When writing the recursive statement, assume the function works as intended for the smaller problems. Trust. (Abstraction! woohoo!)
   - If you don't know what the recursive call needs to be, you can take an educated guess and see what happens.
   - It's often extremely helpful to run line by line through a doctest to test both a tentative solution and your understanding to a problem.
   - When working through a problem, it can be very helpful to visualize recursion in action. For shorter problems, one effective method is to draw a stack of 'boxes,' each representing a recursive call and its result, continuing until you reach the base case. Other linear visualizations can also be effective!

We tend to throw around the term "recursive leap of faith" a lot, and I think that it confuses students. The "recursive leap of faith" is not synonymous with "the recursive call is correct". Rather it's a specific assumption we make while writing a recursive function that the recursive calls we make produce the correct output, even if we're not done writing our function. That is, the function we're writing works even if we're not done writing it. The fact that recursive calls return the correct value in the completed function is a mathematical fact that does not require any faith, so you should not conflate the two. Recursion is not magic; it is math.

The recursive leap of faith is essentially the scaffolding we need to help us build the recursive function. We pour the concrete for the base case and then layer our recursive logic on top of that until we have a sturdy structure, using the leap of faith's scaffolding to help us, as fallible human builders, to figure out the right way for the

pieces to fit together. Once we're done, we can remove the scaffolding, but our tower still stands strong and sturdy.

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)
```

This is a good opportunity to emphasize to your students that the return type of the base case should match the return type of the function (e.g., if the function is expected to return an integer, the base case must also return an integer).

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)
```