HIGHER-ORDER FUNCTIONS & ENVIRONMENT DIAGRAMS Meta

COMPUTER SCIENCE MENTORS 61A

February 3 – February 7, 2025

Example Timeline

- Intros, Ice Breakers, Expectations, Logistics & Brief Intro of CSM [5 min]

  Don't be afraid to spend some time on this! It'll likely make your job in the future easier! Prospective mentees will also get to know how CSM works and doing icebreakers will make it a welcoming experience for all of them but try to keep them kind of short since this is not their official session.

  Also, something we want to do this semester more of is emphasize the conceptual topics and concepts in 61A like functional abstraction and more. So make sure to talk about them regularly throughout all of your teaching sessions and tell junior mentors to also emphasize them as well.

- Environment Diagrams Mini Lecture + Q1 [15 mins]

  1. You can use Q1 as an example for the mini-lecture

  2. A lot of students (even those with significant programming background) get confused by env. diagrams. It's probably worth doing the minilecture even if you have advanced students.

  3. You should address when we need to make a new frame in an environment diagram.

- Higher Order Functions Overview + Reasoning + Q1 [5 mins]
- Problems (You pick which ones): HOF

  1. Foo: has an example of a function which is returned along with return in the middle of function and returning None

  2. Compose: easy example to build skill in writing lambda functions.

  3. WholeSum: Another HOF example and digit manipulation and while loop practice; Skeleton code practice

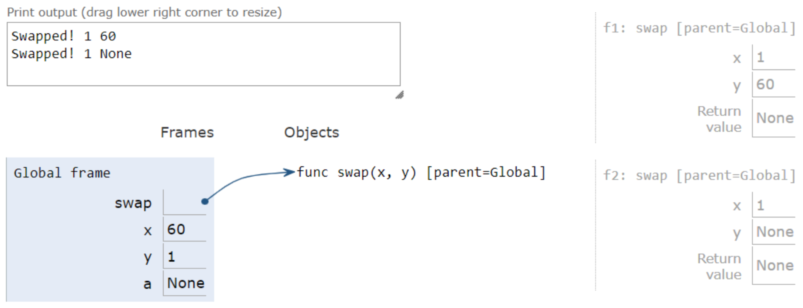
  4. Alternator: HOF example with while loop practice. Implementing whole function from beginning

  5. Curryforever: An example with writing code for a function which takes in a variable number of arguments; the number of arguments that will be taken is input earlier. Skeleton code practice

1. Give the environment diagram and console output that result from running the following code.

```
def swap(x, y):
    x, y = y, x
    return print("Swapped!", x, y)

x, y = 60, 1
a = swap(x, y)
swap(a, y)
```


Print output (drag lower right corner to resize)

```
Swapped! 1 60
Swapped! 1 None
```

https://tinyurl.com/y68m6qdj

Suggested Time: 5 min; Difficulty: Medium

- Go over the order in which call expressions are evaluated and emphasize that a lot since that is a fundamental of this class.
- Remind them that for user-defined functions, both lambda functions and functions created with def keyword, that parent is the function it is defined in.
- Another tip we find useful is creating a sort of graph to graph how the functions are being called where vertices are fxn(arguments) and directed edges start from the vertex containing fxn where fxn where edge ends at is called in.
- This question stresses variables in different scopes.

  Show difference between `x` and `y` in both global and local frames.

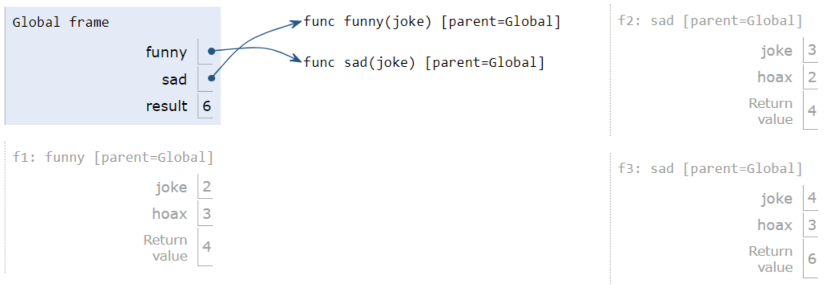
  Also note to students that a call to `swap(x, y)` will not actually swap the values of `x` and `y` in the frame where it is called.
- It might also be good to recap what `x, y = y, x` does in python – ensure students know that this is a special feature of python and that switching happens in 1 line, by order of how the values are listed. Here, the expressions on the right hand side of the assignment operator are evaluated first from left to right, then the assignment happens to the names on the left of the assignment operator from left to right.

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

```
def funny(joke):
    hoax = joke + 1
    return funny(hoax)

def sad(joke):
    hoax = joke - 1
    return hoax + hoax

funny, sad = sad, funny
result = funny(sad(2))
```



https://tinyurl.com/y5lc4fez

Suggested Time: 10 min; Difficulty: Medium

- Make sure that the students understand how Python looks for a value of a variable, from local (to parent(s)) to global.
- Make sure your students understand the difference between an intrinsic name and a bound name

  Intrinsic: For user defined functions, this intrinsic name is the name used in the **def** statement

  Bound: Names of variables that point to the function object. A function can have many bound names, and the bound names of a function can often change.

1. What are higher-order functions? Why and where do we use lambda and higher-order functions? Can you give a practical example of where we would use a HOF?

   Higher-order functions are functions that does at least one of the following: take at least one or more functions as arguments and returns a function. In practice, we use lambda functions to pass code as data in a concise manner. One specific example to illustrate the use of lambdas is the optional `key` parameter for **`min`** and **`max`** functions. Lambda functions can be passed as arguments to higher-order functions. Higher order functions serve as a tool of abstraction, allowing us to simplify repeated actions into one function that we can use over and over again. Students can have varying answers for practice uses of HOFs, though here are some suggestions for the average student coming across this worksheet:

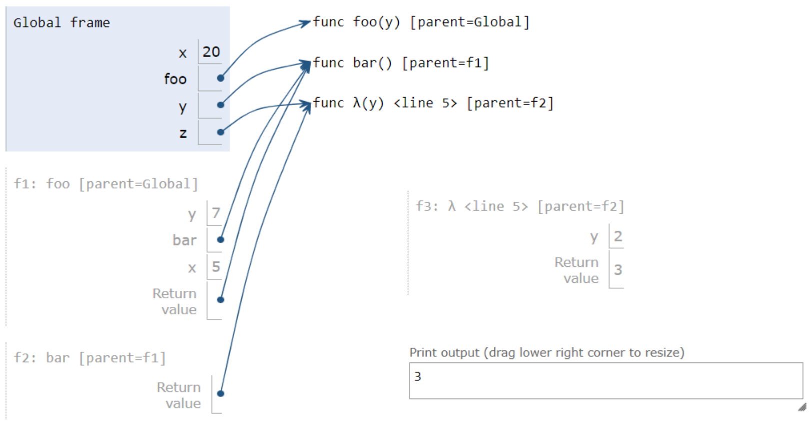
   - Our method signature is composed of one parameter, but we wish to use a higher order function with more parameters to abstract extra steps.
   - When our function is long and complex; easier to read code when it's organized into several different higher order functions.

   Suggested Time: 5 mins Try to give examples when explaining HOFs. Some examples are linked at this link: `https://tinyurl.com/5fsd3h8b`.

2. Give the environment diagram and console output that result from running the following code.

```
x = 20
def foo(y):
    x = 5
    if y == 5:
        return lambda y: x + y
    else:
        print('hello!')

y = foo(5)
x = y(7)
z = foo(7)
```



Print output (drag lower right corner to resize)

```
3
```

`https://tinyurl.com/4dkbpnyc`

Suggested Time: 7 mins; Difficulty: Easy

- Emphasize that if line with return statement is reached, then expression after return keyword will be evaluated and that value is then returned to the environment/frame in which the function returning the value was called in.
- Also, emphasize that if no return statement is encountered while calling a function with certain arguments then the return value is None.

3. Implement `compose`.

```
def compose(f, g):
    """
    >>> a = compose(lambda x: x * x, lambda x: x + 4)
    >>> a(2)
    36
    """

    return lambda x: f(g(x))
```

Suggested Time: 5 mins; Difficulty: Easy

4. Write a function, `whole_sum`, which takes in an integer, `n`. It returns another function which takes in an integer, and returns `True` if the digits of that integer sum to `n` and `False` otherwise.

```
def whole_sum(n):
    """
    >>> whole_sum(21)(777)
    True
    >>> whole_sum(142)(10010101010)
    False
    """
    def check(x):

        ___________________

        while _____________:

            last = __________________

            _________________________

            _________________________

        return __________________

    return _________________
```

```
def whole_sum(n):
    def check(x):
        total = 0
        while x > 0:
            last = x % 10
            x = x // 10
            total += last
        return total == n
    return check
```

Suggested Time: 8 Mins; Difficulty: Medium

- Remind your students that for HOFs, you must **`return`** the inner function (ie we must **`return`** `check` to use it).
- Also depending on the skill level of students in your section, a recap of digit manipulation may be needed (ie x // 10, x % 10, etc.)

5. Implement `make_alternator` which takes in two functions and outputs a function. The returned function takes in a number `x` and prints out all the numbers from 1 to `x`, applying `f` to the odd numbers and applying `g` to the even numbers before printing.

```
def make_alternator(f, g):
    """
    >>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
    >>> a(5)
    1
    6
    9
    8
    25
    """


    def alternator(x):
        i = 1
        while i <= x:
            if i % 2 == 1:
                print(f(i))
            else:
                print(g(i))
            i += 1
    return alternator
```

Teaching Tips

- Again, walk students through each iteration from 1 to x, and show how each of the two functions `f,g` alternate on incrementing inputs.
- Remember the general structure needed whenever a function must return a function.

Suggested Time: 8 mins; Difficulty: Medium

- Walk students through each iteration from 1 to x, and show how each of the two functions `f,g` alternate on incrementing inputs.
- Remember the general structure needed whenever a function must return a function.

6. Write a function, `curry_forever`, which takes in a two-argument function, `f`, and an integer, `arg_num`. It returns another function that allows us to enter arg_num amount of numbers into f one by one.

```
def curry_forever(f, arg_num, base=0):
    """
    >>> g = curry_forever(lambda x, y: x + y, 4)
    >>> g(1)(2)(3)(4) # 1 + 2 + 3 + 4
    10
    """

    def helper(arg_num, amt):

         if arg_num == 0:

            ______________________________________________

         return ________________________________________

    ____________________________________________________
```

```
def curry_forever(f, arg_num, base=0):
    def helper(arg_num, amt):
        if arg_num == 0:
                return amt
        return lambda x: helper(arg_num - 1, f(amt, x))
    return helper(arg_num, base)
```

Suggested Time: 15 mins; Difficulty: Medium

7. Students might be confused on how to build a function which takes in a variable number of arguments. If they are stuck, give them some hints on how to approach in general questions with skeleton code.