Environment Diagrams and Higher Order Functions Meta

Computer Science Mentors 61A

September 7–September 9, 2022

Recommended Timeline

Times do not add to one hour because it is not expected that any mentor will get through all questions in a single hour.

- Environment Diagrams Mini-Lec + New Frame Written Question - 12 mins
- Swap - 8 mins
- Joke - 7 mins
- Higher Order Functions Mini-Lec + Written Question - 3 mins
- Foobar or Apple (Q2 or Q3): these two problems are pretty similar - 10 mins
- xyz - 8 mins
- whole_sum or mystery (Q5 or Q6+7) - if there isn't enough time, pick depending and how students feel; 5 is a pretty standard HOF question that involves digit manipulation, 6+7 for some more practice with lambdas - 10 mins

1 Environment Diagrams

1. When do we make a new frame in an environment diagram?

   We make a new frame in an environment diagram when calling a user-defined function, or when we are applying the operator to the operand(s). This occurs after both the operator and operand(s) are evaluated.

   New frames are *not* created when a function is defined, only when it is called.

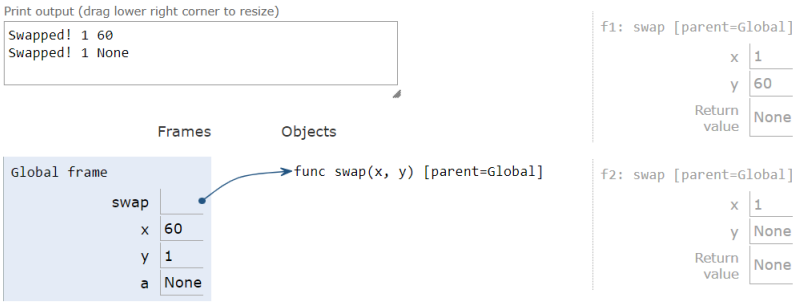
   This is meant to be a very quick warm-up question with the goals of 1) cementing the rules of evaluation and 2) emphasizing that frames are opened only when functions are called, not when they are defined.

Vibha Tantry and Jordan Schwartz, with
Gabe Classon, Aditya Balasubramanian, Oliver Yu, Hanze Tan, Gene Pan, Carolyn Wang, Raman Varma, Noor Haq, Angela Zhang, Abigail Brooks, Ted Kim, Aiden Joshua Vehemente, Marie Chorpita, Ashley Chiu, Bill Hu, Omar Yu, Angel Alberto Aldaco, Andrew Park, Chase Clements, Grace Yi, Preshtha Garg, Laksith Venkatesh Prabu, Aditya Murali, Irene Yang, Andy Chen, and Hailey Park

2. 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)
```



https://tinyurl.com/y68m6qdj

This question emphasizes the separation of variables into different scopes. The fact that `x` and `y` are used in both global and local frames is meant to demonstrate that the same symbol can mean different things in different contexts, and that different frames have domain over their own namespace and—for the most part—cannot change the namespace of other frames. The printed output ("Swapped! ...") and name of the function (`swap`) are a bit misleading in this respect because a call to `swap(x, y)` will not actually swap the values of `x` and `y` in the frame where it is called. (That is, `x` and `y` still have the same values in the global frame after `swap` is called on them.) The goal is that students will be surprised/confused by this behavior, and that they will learn from this mental challenge.

Some students might have done variable swaps in other languages. If students are confused by this, make sure to explain why `x, y = y, x` works in Python.

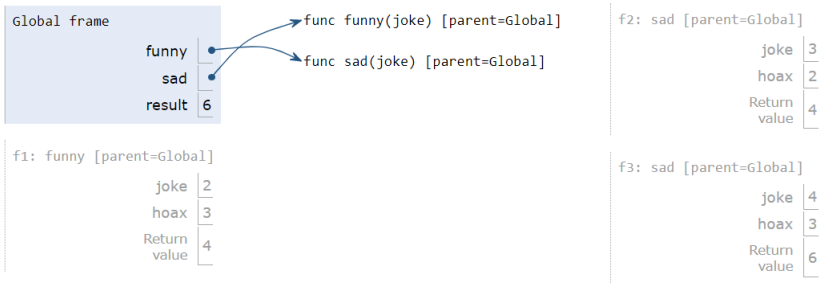
It may also be judicious to explain the difference between printing and returning, which can be confusing to some students.

3. 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

The primary purpose of this problem is to teach variable lookup rules and the difference between intrinsic and bound names of functions.

Make sure that the students understand how Python looks for a value of a variable, from local (to parent(s)) to global.

As a reminder, the intrinsic name of a function is the name that belongs to a particular function object. (For a function `f`, this name can be accessed and changed through `f.__name__`, though telling your students this might just confuse them.) For user defined functions, this intrinsic name is the name used in the **`def`** statement (lambda functions have no intrinsic name). Functions cannot necessarily be referred to by their intrinsic names in code. Because it is associated with the function object, the intrinsic name appear on the right-hand side of an environment diagram, in the "objects" column.

The bound name(s) of a function, on the other hand, are the 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.

An analogy I like to use for teaching this is beanie babies. Beanie babies come with a little tag attached that gives the canonical name of the plushie, but most people come up with their own names for their beanie babies and largely ignore this name. For example, when I acquired a beanie baby bear, its tag name was "Glory", but I called it "Aces". The previous owner called it "Washington". Depending on the context, different people can call the bear different things, but the name on the tag, which is physically attached to the bear, does not change. Similarly, different frames can have different bound names for a function, but the intrinsic name is attached to the function object and has permanence.

When teaching this problem, also remember to evaluate the functions on the RHS first, then assign to variables on the LHS.

2 Higher-Order Functions

1. Why and where do we use lambda and higher-order functions?

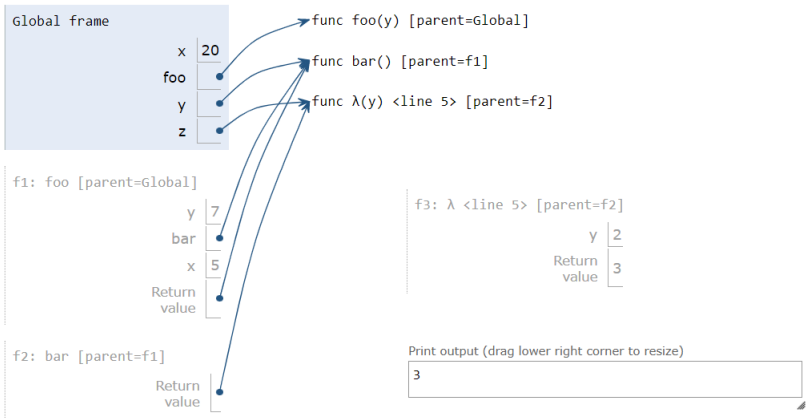
   In practice, we use lambda functions to pass code as data in a concise manner. One specific example to illustrate the use of lambdas is the optional `key` parameter for **`min`** and **`max`** functions. Higher order functions serve as a tool of abstraction, allowing us to simplify repeated actions into one function that we can use over and over again, also referred to as currying.

   Feel free to talk about any other practical examples of lambdas or HOFs that you know about!

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

```
x = 20
def foo(y):
    x = 5
    def bar():
        return lambda y: x - y
    return bar

y = foo(7)
z = y()
print(z(2))
```



https://tinyurl.com/yxfcvxxa

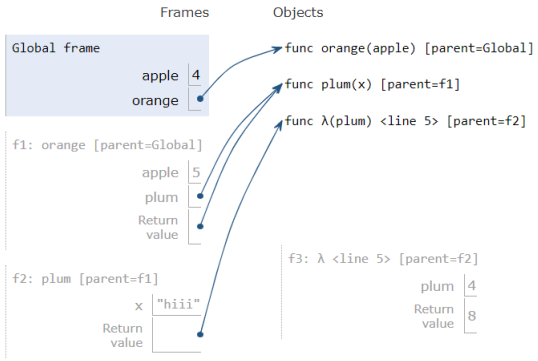
Make sure that your students understand the process of looking up the value of a variable in Python.

It's probably not super important for you to do both `foobar` and `apple`, as these questions are pretty similar.

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

```
apple = 4
def orange(apple):
    apple = 5
    def plum(x):
        return lambda plum: plum * 2
    return plum

orange(apple)("hiii")(4)
```



https://tinyurl.com/y5lo34xb

It's probably not super important for you to do both `foobar` and `apple`, as these questions are pretty similar.

4. Fill in the blanks (*without using any numbers in the first blank*) such that the entire expression evaluates to `9`.

```
(lambda x: lambda y: lambda: y(x))(3)(lambda z: z*z)()
```

Notice the arguments passed into each function call. Use these to help students break down the nested lambdas (i.e. the variable y will be assigned to the lambda function with parameter z). For tricky skeleton problems like this, I like to tell students that they are "detectives" and that their mission is to 1) use the available evidence to come up with a hypothesis for how a function is supposed to work and 2) test that hypothesis by attempting to implement the function, returning to step 1 if they find their hypothesis to be incorrect.

5. 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
```

Great time to emphasize that whenever you write a HOF, you must return the inner function in order to use it properly (i.e. in order to use function `check` we must return on it the last line). This is quite an "easy" thing to check and it's a relatively accessible point on exams to students who are familiar with it.

We suggest that you go through this problem step by step in order to solidify digit manipulation concepts (i.e. x // 10, x % 10).

6. Write a higher-order function that passes the following doctests.

   Challenge: Write the function body in one line.

```
def mystery(f, x):
    """
    >>> from operator import add, mul
    >>> a = mystery(add, 3)
    >>> a(4) # add(3, 4)
    7
    >>> a(12)
    15
    >>> b = mystery(mul, 5)
    >>> b(7) # mul(5, 7)
    35
    >>> b(1)
    5
    >>> c = mystery(lambda x, y: x * x + y, 4)
    >>> c(5)
    21
    >>> c(7)
    23
    """


    def helper(y):
        return f(x, y)
    return helper
```

   Challenge solution:

```
    return lambda y : f(x, y)
```

   Using doctests to understand how a function should work is a fundamental part of CS 61A. The goal of this question is to force students to exercise that muscle by removing any other description of `mystery`.

   The advice noted before about acting as a "detective" applies again to this problem.

7. What would Python display?

```
>>> foo = mystery(lambda a, b: a(b), lambda c: 5 + square(c))
>>> foo(-2)


9
```