Higher-Order Functions, Environment Diagrams, & Functional Abstraction Solutions

Computer Science Mentors 61A

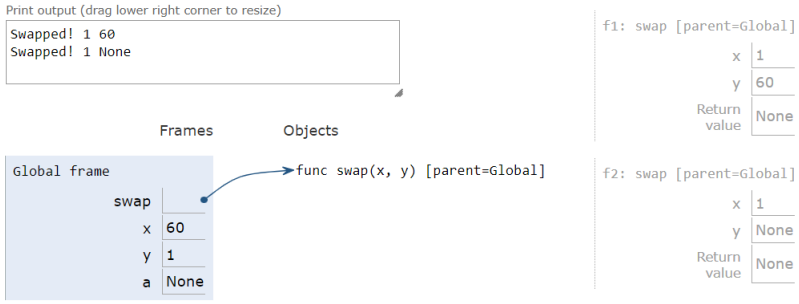
February 3 – February 7, 2025

1 Environment Diagrams

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)
```



https://tinyurl.com/y68m6qdj

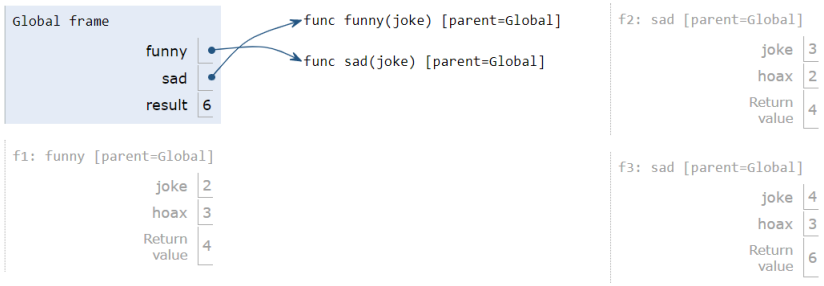
Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

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

2 Higher-Order Functions

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?

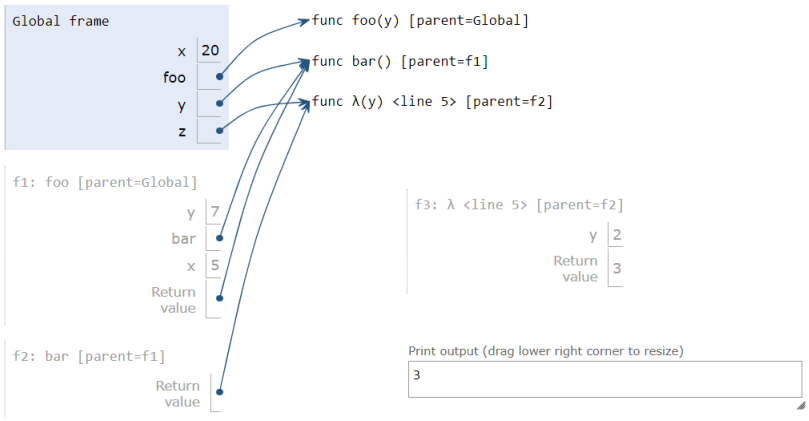
   Higher-order functions are functions that does at least one of the following: take at least one or more functions as arguments and returns a function. In practice, we use lambda functions to pass code as data in a concise manner. One specific example to illustrate the use of lambdas is the optional `key` 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.

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)
```



https://tinyurl.com/4dkbpnyc

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))
```

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
```

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
```