Higher Order Functions & Environment Diagrams

• Environment Diagrams

- https://docs.google.com/presentation/d/1i1Ojc8MJpNh195O-sf6ZDAf0urRYygdv0OZ7EYUWPYI/edit#slide=id.p
- http://albertwu.org/cs61a/notes/environments.html

① Always label parent frame (Frame function was defined in)
↳ all frames other than global have a parent frame

② Be careful to look in correct scope for variable lookups
↳ if you can't find a variable in the current frame, look in that frame's parent

• Lambda Functions

- one line expressions that evaluate to functions (not evaluated until lambda is called)
  ↳ create function values "on the fly" (no intermeadiate function name)

```
lambda    x      :    (x * x) + 3
```
"A function that takes x and returns (x * x) + 3"

- terminal example

```
>> a = lambda x: x + x
>> a
<function <lambda> at 0xf3f904>
>> a(3)
6
```

• Higher Order Functions

- a function that returns a function or takes a function as an input
  ↳ lots of practice with this in Hog!!
- lexical scope → locally defined functions also have access to the name bindings in the scope in which they are defined (parent frame)

Practice Problems from Discussion wksht

Q2: Make Keeper

1.2 **Tutorial:** Write a function similar to `keep_ints` like before, but now it takes in a number `n` and returns a function that has one parameter `cond`. The returned function prints out numbers from 1 to `n` where calling `cond` on that number returns `True`.

← HOF (returns a function)

← good strategy is to start by writing code for this

```
def make_keeper(n):
    """Returns a function which takes one parameter cond and prints out
    all integers 1..i..n where calling cond(i) returns True.

    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> make_keeper(5)(is_even)
    2
    4
    """
```

remember this returns a function that takes another function as a parameter

① individual work time → 5 min.
② split into breakout rooms to discuss if want (open rooms) → last 3 min.

Guided walkthrough

① HOF general structure

```
def f(x):
    def g(y):
        ⋮
    return g
```

② How can we iterate through all the numbers from 1 to n?
counter variable & while loop

③ How can I put these pieces together?
think about what our return value should be / look at given doctests!

SOLUTION:

```
def make_keeper(n):
    def all_true(cond):
        i = 1
        while (i <= n):
            if (cond(i) == True):
                print(i)
            i = i + 1
    return all_true
```

body of function → just print all $i \in \{1, \ldots, n\}$ that satisfy the condition cond(i) = True

← return a function that will then do all the work

Q5 : HOF Diagram Practice

1.5 **Tutorial:** Draw the environment diagram that results from executing the code below.

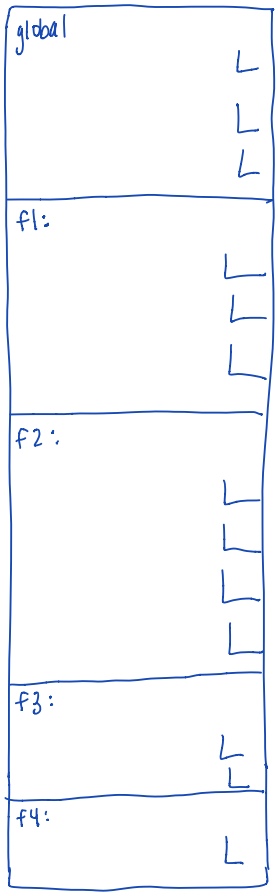
```
1  n = 7
2
3  def f(x):
4      n = 8
5      return x + 1
6
7  def g(x):
8      n = 9
9      def h():
10         return x + 1
11     return h
12
13 def f(f, x):
14     return f(x + n)
15
16 f = f(g, n)
17 g = (lambda y: y())(f)
```

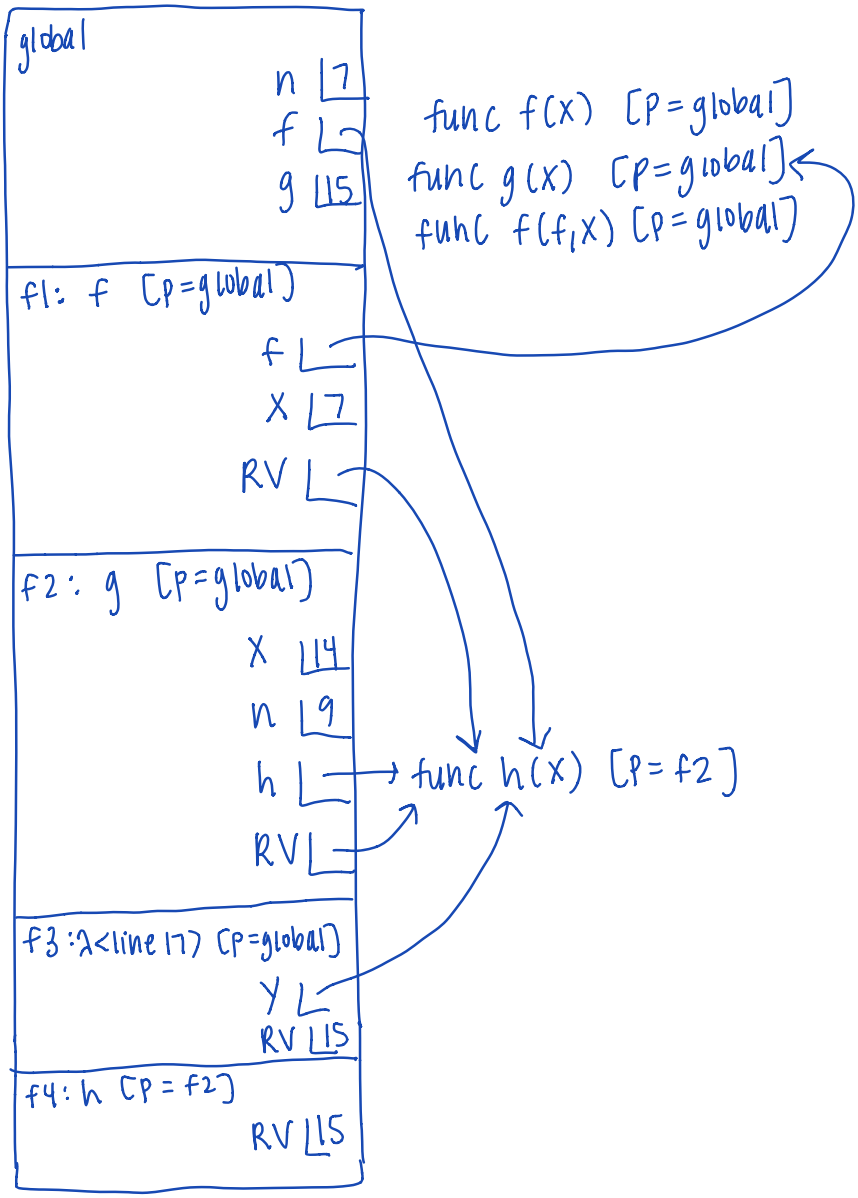
Please type in chat which option you prefer?

(#1) Attempt on your own (≈ 3-4 mins) and then walk through solutions together

(#2) Directed walk-through written first and then using pythontutor (screen share)
↳ collaboratively guessing/sharing ideas as we go


```
1  n = 7
2
3  def f(x):
4      n = 8
5      return x + 1
6
7  def g(x):
8      n = 9
9      def h():
10         return x + 1
11     return h
12
13 def f(f, x):
14     return f(x + n)
15
16 f = f(g, n)
17 g = (lambda y: y())(f)
```


global
n 7
f
g 15
func f(x) [P=global]
func g(x) [P=global]
func f(f,x) [P=global]
f1: f [P=global]
f
X 7
RV
f2: g [P=global]
X 14
n 9
h
func h(x) [P=f2]
RV
f3: λ<line 17> [P=global]
y
RV 15
f4: h [P=f2]
RV 15

Q9: Print N

returning a function (HOF)

1.8 **Tutorial:** Write a function `print_n` that can take in an integer `n` and returns a repeatable print function that can print the next `n` parameters. After the nth parameter, it just prints "done".

need some way to keep track of which one we are currently on

some "final" condition

```
def print_n(n):
    """
    >>> f = print_n(2)
    >>> f = f("hi")
    hi
    >>> f = f("hello")
    hello
    >>> f = f("bye")
    done
    >>> g = print_n(1)
    >>> g("first")("second")("third")
    first
    done
    done
    <function inner_print>
    """
    def inner_print(x):
        if _________________________
            print("done")
        else:
            print(x)
        return _____________________
    return _________________________
```

integer input

returns a function

now f is the print function
↳ first parameter ✓

second parameter ✓

third parameter => should print "done" since initial parameter n = 2

- (2 min) individual work / ideas for approach
- (5 min) discuss / walk-through

SOLUTION:

```
def inner_print(x):
    if n <= 0 _____________________
        print("done")
    else:
        print(x)
    return print_n(n-1)
return inner_print
```

← the "easy check": when do we want to return "done"? (has to be <= because we see in the second doctest that all inputs after nth input should print ("done"))

← this is the self-reference part! way to have repeatable print function & to update/keep track of n properly!

← general structure of HOFs (more intuitively, we need some way to reach the inner function but on the first pass through print_n we only define inner & then return out)
↳ draw out env. diagram (pythontutor) to see this!

Q7: Make Keeper Redux

Q7: (Tutorial) Warm Up: Make Keeper Redux

These exercises are meant to help refresh your memory of topics covered in lecture and/or lab this week before tackling more challenging problems.

In this question, we will explore the execution of a self-reference function, `make_keeper_redux`, based off Question 2, `make_keeper`. The function `make_keeper_redux` is similar to `make_keeper`, but now the returned function also returns **another function** with the same behavior. Feel free to paste and modify your code for `make_keeper` below.

(Hint: you only need to add one line to your `make_keeper` solution. What is currently missing from `make_keeper_redux`?)

```
def make_keeper_redux(n):
    """Returns a function. This function takes one parameter <cond> and prints out
       all integers 1..i..n where calling cond(i) returns True. The returned
       function returns another function with the exact same behavior.

        >>> def multiple_of_4(x):
        ...     return x % 4 == 0
        >>> def ends_with_1(x):
        ...     return x % 10 == 1
        >>> k = make_keeper_redux(11)(multiple_of_4)
        4
        8
        >>> k = k(ends_with_1)
        1
        11
        >>> k
        <function do_keep>
    """
    # Paste your code for make_keeper here!


```

```
def all_True(cond):
    i = 1
    while (i <= n):
        if (cond(i) == True):
            print(i)
        i = i + 1
    return all_True
```

copied from make_keeper

↳what can we do to make 'inner' function also return a function that essentially does this all over again?

SOLUTION:

```
def make_keeper_redux(n):
    def all_True(cond):
        i = 1
        while (i <= n):
            if (cond(i) == True):
                print(i)
            i = i + 1
        return make_keeper_redux(n)
    return all_True
```