

# Discussion 2: Higher-Order Functions, Self Reference, Lambda Expressions

This is an online worksheet that you can work on during discussions. Your work is not graded and you do not need to submit anything. The last section of most worksheets is Exam Prep, which will typically only be taught by your TA if you are in an Exam Prep section. You are of course more than welcome to work on Exam Prep problems on your own.

## Higher Order Functions

A **higher order function** (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. For example, the function `compose1` below takes in two functions as arguments and returns a function that is the composition of the two arguments.

```
def compose1(f, g):  
    def h(x):  
        return f(g(x))  
    return h
```

HOFs are powerful abstraction tools that allow us to express certain general patterns as named concepts in our programs.

## A Note on Lambda Expressions

A lambda expression evaluates to a function, called a lambda function. For example, `lambda y: x + y` is a lambda expression, and can be read as “a function that takes in one parameter `y` and returns `x + y`.”

A lambda expression by itself evaluates to a function but does not bind it to a name. Also note that the return expression of this function is not evaluated until the lambda is called. This is similar to how defining a new function using a `def` statement does not execute the function’s body until it is later called.

```
>>> what = lambda x : x + 5  
>>> what  
<function <lambda> at 0xf3f490>
```

Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions. In the example below, `(lambda y: y + 5)` is the operator and `4` is the operand.

```
>>> (lambda y: y + 5)(4)  
9  
>>> (lambda f, x: f(x))(lambda y: y + 1, 10)  
11
```

## Q1: Keep Ints

Write a function that takes in a function `cond` and a number `n` and prints numbers from 1 to `n` where calling `cond` on that number returns `True`.

```
1 def keep_ints(cond, n):
2     """Print out all integers 1..i..n where cond(i) is true
3     >>> def is_even(x):
4         ...     # Even numbers have remainder 0 when divided by 2.
5         ...     return x % 2 == 0
6     >>> keep_ints(is_even, 5)
7     2
8     4
9     """
10    """ YOUR CODE HERE """
11
12
```

## Q2: Make Keeper

Write a function similar to `keep_ints` like in Question 1, 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`.

```

1  def make_keeper(n):
2      """Returns a function which takes one parameter cond and prints out all integers 1..i..n
3          >>> def is_even(x):
4              ...     # Even numbers have remainder 0 when divided by 2.
5              ...     return x % 2 == 0
6          >>> make_keeper(5)(is_even)
7              2
8              4
9          """
10     """*** YOUR CODE HERE ***"""
11
12

```

## HOFs in Environment Diagrams

Recall that an **environment diagram** keeps track of all the variables that have been defined and the values they are bound to. However, values are not necessarily only integers and strings. Environment diagrams can model more complex programs that utilize higher order functions.

Server error! Your code might have an INFINITE LOOP or be running for too long.

The server may also be OVERLOADED. Or you're behind a FIREWALL that blocks access.

Try again later. This site is free with NO technical support. [#UnknownServerError]

(see [UNSUPPORTED FEATURES](#))

Lambdas are represented similarly to functions in environment diagrams, but since they lack intrinsic names, the lambda symbol ( $\lambda$ ) is used instead. If there are multiple lambda statements in the code, we may index them  $\lambda_1$ ,  $\lambda_2$ , etc.

The parent of any function (including lambdas) is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call `add_two` (which is really the lambda function), we need to know what `x` is in order to compute `x + y`. Since `x` is not in the frame `f2`, we look at the frame's parent, which is `f1`. There, we find `x` is bound to 2.

As illustrated above, higher order functions that return a function have their return value represented with a pointer to the function object.

## Currying

One important application of HOFs is converting a function that takes multiple arguments into a chain of functions that each take a single argument. This is known as **currying**. For example, the function below converts the `pow` function into its curried form:

```
>>> def curried_pow(x):  
    def h(y):  
        return pow(x, y)  
    return h  
  
>>> curried_pow(2)(3)  
8
```

### Q3: Curry2 Diagram

Draw the environment diagram that results from executing the code below.

```
def curry2(h):  
    def f(x):  
        def g(y):  
            return h(x, y)  
        return g  
    return f  
  
make_adder = curry2(lambda x, y: x + y)  
add_three = make_adder(3)  
add_four = make_adder(4)  
five = add_three(2)
```

**Global frame**

		●
		●
		●
		●
		●

**f1:**  [parent=]

		●
		●
Return value	<input type="text"/>	●

**f2:**  [parent=]

		●
		●
Return value	<input type="text"/>	●

**f3:**  [parent=]

		●
		●
Return value	<input type="text"/>	●


**f4:**  [parent=]

		●
		●
Return value	<input type="text"/>	●

**f5:**  [parent=]

		●
		●
Return value	<input type="text"/>	●

**Objects**

▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
	

## Q4: Curry2 Lambda

Write `curry2` as a lambda function.

```
1  "*** YOUR CODE HERE ***"  
2  
3
```

## Q5: HOF Diagram Practice

Draw the environment diagram that results from executing the code below.

```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)
g = (lambda y: y()(f))
```

Global frame

		●
		●
		●

f1:  [parent=]

		●
		●
		●
Return value	<input type="text"/>	●

f2:  [parent=]

		●
		●
		●
Return value	<input type="text"/>	●

f3:  [parent=]

		●
		●
Return value	<input type="text"/>	●

f4:  [parent=]

		●
		●
Return value	<input type="text"/>	●

Objects

▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>





# Self Reference

Self-reference refers to a particular design of HOF, where a function eventually returns itself. In particular, a self-referencing function will not return a function call, but rather the function object itself. As an example, take a look at the `print_all` function:

```
def print_all(x):  
    print(x)  
    return print_all
```

Self-referencing functions will oftentimes employ helper functions that reference the outer function, such as the example below, `print_sums`.

```
def print_sums(n):  
    print(n)  
    def next_sum(k):  
        return print_sums(n + k)  
    return next_sum
```

A call to `print_sums` returns `next_sum`. A call to `next_sum` will return the result of calling `print_sums` which will, in turn, return another function `next_sum`. This type of pattern is common in self-referencing functions.

A Note on Recursion: This differs from recursion because typically each new call returns a new function rather than a function call. We have not yet covered recursion so don't worry too much about what this means!

## Q6: Make Keeper Redux

In this question, we will explore the execution of a self-reference function, `make_keeper_redux`, based off Question 2 (</~cs61a/su21/disc/disc02/#q2>), `make_keeper`. The function `make_keeper_redux` is similar to `make_keeper`, but now the function returned by `make_keeper_redux` should be self-referential—i.e., the returned function should return a function with the same behavior as `make_keeper_redux`. 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`?)

```

1  def make_keeper_redux(n):
2      """Returns a function. This function takes one parameter <cond> and prints out
3          all integers 1..i..n where calling cond(i) returns True. The returned
4          function returns another function with the exact same behavior.
5
6          >>> def multiple_of_4(x):
7              ...     return x % 4 == 0
8          >>> def ends_with_1(x):
9              ...     return x % 10 == 1
10         >>> k = make_keeper_redux(11)(multiple_of_4)
11         4
12         8
13         >>> k = k(ends_with_1)
14         1
15         11
16         >>> k
17         <function do_keep>
18         """
19         # Paste your code for make_keeper here!
20
21

```

## Q7: Print Delayed

Write a function `print_delayed` that delays printing its argument until the next function call. `print_delayed` takes in an argument `x` and returns a new function `delay_print`. When `delay_print` is called, it prints out `x` and returns another `delay_print`.

```
1 def print_delayed(x):
2     """Return a new function. This new function, when called, will print out x and return and
3     >>> f = print_delayed(1)
4     >>> f = f(2)
5     1
6     >>> f = f(3)
7     2
8     >>> f = f(4)(5)
9     3
10    4
11    >>> f("hi") # a function is returned
12    5
13    <function delay_print>
14    """
15    def delay_print(y):
16        _____
17        return _____
18    return delay_print
19
```

## Q8: Print N

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".

```
1  def print_n(n):
2      """
3      >>> f = print_n(2)
4      >>> f = f("hi")
5      hi
6      >>> f = f("hello")
7      hello
8      >>> f = f("bye")
9      done
10     >>> g = print_n(1)
11     >>> g("first")("second")("third")
12     first
13     done
14     done
15     <function inner_print>
16     """
17     def inner_print(x):
18         if _____:
19             print("done")
20         else:
21             print(x)
22         return _____
23     return _____
24
25
```

## Exam prep

---

We recommending reading sections 1.1-1.6 from the textbook for these problems.

Test your work! For example, for `match_k`, you can type `test(match_k)` in the python interpreter you get once you click Run in 61A Code to verify if you pass the doctests or not.

## Q9: YY Diagram

Draw the environment diagram that results from executing the code below.

Tip: Using the `+` operator with two strings results in the second string being appended to the first. For example `"C" + "S"` concatenates the two strings into one string `"CS"`.

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```

Global frame

		●
		●

f1:  [parent=]

		●
		●
Return value	<input type="text"/>	●

f2:  [parent=]

		●
		●
Return value	<input type="text"/>	●

f3:  [parent=]

		●
		●
Return value	<input type="text"/>	●

f4:  [parent=]

		●
		●
Return value	<input type="text"/>	●

Objects

▶	<input type="text"/>
▶	<input type="text"/>
▶	<input type="text"/>



Video walkthrough (<https://www.youtube.com/watch?v=MLRfJaGBeAY&feature=youtu.be>)

## Q10: Match Maker

Implement `match_k`, which takes in an integer `k` and returns a function that takes in a variable `x` and returns `True` if all the digits in `x` that are `k` apart are the same.

For example, `match_k(2)` returns a one argument function that takes in `x` and checks if digits that are 2 away in `x` are the same.

`match_k(2)(1010)` has the value of `x = 1010` and digits 1, 0, 1, 0 going from left to right. `1 == 1` and `0 == 0`, so the `match_k(2)(1010)` results in `True`.

`match_k(2)(2010)` has the value of `x = 2010` and digits 2, 0, 1, 0 going from left to right. `2 != 1` and `0 == 0`, so the `match_k(2)(2010)` results in `False`.

**RESTRICTION:** You may not use strings or indexing for this problem.

**IMPORTANT:** You do not have to use all the lines, one staff solution does not use the line directly above the while loop.

**HINT:** Floor dividing by powers of 10 gets rid of the rightmost digits.

```

1  def match_k(k):
2      """ Return a function that checks if digits k apart match
3
4      >>> match_k(2)(1010)
5      True
6      >>> match_k(2)(2010)
7      False
8      >>> match_k(1)(1010)
9      False
10     >>> match_k(1)(1)
11     True
12     >>> match_k(1)(2111111111111111)
13     False
14     >>> match_k(3)(123123)
15     True
16     >>> match_k(2)(123123)
17     False
18     """
19     _____
20
21     while _____:
22         if _____:
23             return _____
24         _____
25     _____
26     _____
27
28     ~

```



## Q11: My Last Three Brain Cells

A *k*-memory function takes in a single input, prints whether that input was seen exactly *k* function calls ago, and returns a new *k*-memory function. For example, a 2-memory function will display “Found” if its input was seen exactly two function calls ago, and otherwise will display “Not found”.

Implement `three_memory`, which is a three-memory function. You may assume that the value `None` is never given as an input to your function, and that in the first two function calls the function will display “Not found” for any valid inputs given.

```

1  def three_memory(n):
2      """
3      >>> f = three_memory('first')
4      >>> f = f('first')
5      Not found
6      >>> f = f('second')
7      Not found
8      >>> f = f('third')
9      Not found
10     >>> f = f('second') # 'second' was not input three calls ago
11     Not found
12     >>> f = f('second') # 'second' was input three calls ago
13     Found
14     >>> f = f('third') # 'third' was input three calls ago
15     Found
16     >>> f = f('third') # 'third' was not input three calls ago
17     Not found
18     """
19     def f(x, y, z):
20         def g(i):
21             if _____:
22                 _____
23             else:
24                 _____
25             return _____
26         return _____
27     return f(None, None, n)
28 
```

## Q12: Natural Chainz

For this problem, a `chain_function` is a higher order function that repeatedly accepts natural numbers (positive integers). The first number that is passed into the function that `chain_function` returns initializes a natural chain, which we define as a consecutive sequence of increasing natural numbers (i.e., 1, 2, 3). A natural chain breaks when the next input differs from the expected value of the sequence. For example, the sequence (1, 2, 3, 5) is broken because it is missing a 4.

Implement the `chain_function` so that it prints out the value of the expected number at each chain break as well as the number of chain breaks seen so far, including the current chain break. Each time the chain breaks, the chain restarts at the most recently input number.

For example, the sequence (1, 2, 3, 5, 6) would only print 4 and 1. We print 4 because there is a missing 4, and we print 1 because the 4 is the first number to break the chain. The 5 broke the chain and restarted the chain, so from here on out we expect to see numbers increasingly linearly from 5. See the doctests for more examples. You may assume that the higher-order function is never given numbers  $\leq 0$ .

**IMPORTANT:** For this problem, the starter code template is just a suggestion. You are welcome to add/delete /modify the starter code template, or even write your own solution that doesn't use the starter code at all.

```

1  def chain_function():
2      """
3      >>> tester = chain_function()
4      >>> x = tester(1)(2)(4)(5) # Expected 3 but got 4, so print 3. 1st chain break, so print 2
5      3 1
6      >>> x = x(2) # 6 should've followed 5 from above, so print 6. 2nd chain break, so print 2
7      6 2
8      >>> x = x(8) # The chain restarted at 2 from the previous line, but we got 8. 3rd chain break
9      3 3
10     >>> x = x(3)(4)(5) # Chain restarted at 8 in the previous line, but we got 3 instead. 4th
11     9 4
12     >>> x = x(9) # Similar logic to the above line
13     6 5
14     >>> x = x(10) # Nothing is printed because 10 follows 9.
15     >>> y = tester(4)(5)(8) # New chain, starting at 4, break at 6, first chain break
16     6 1
17     >>> y = y(2)(3)(10) # Chain expected 9 next, and 4 after 10. Break 2 and 3.
18     9 2
19     4 3
20     """
21     def g(x, y):
22         def h(n):
23             if _____:
24                 return _____
25             else:
26                 _____
27         return _____
28     return _____
29
30 
```

