

1 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.

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

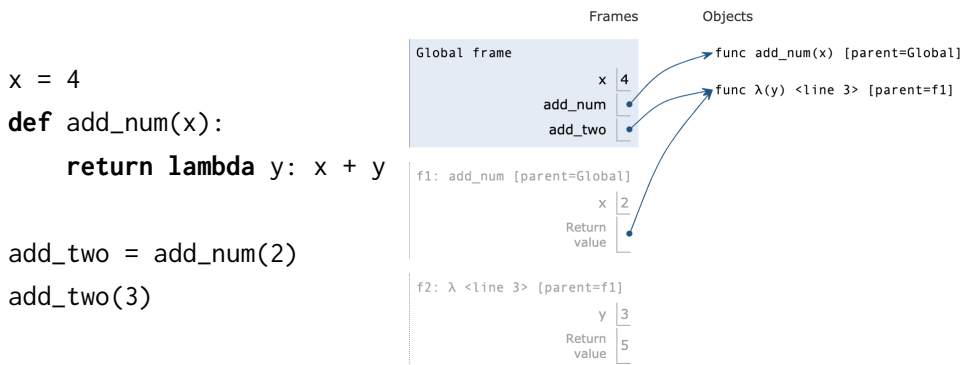
Currying

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
```

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.



Lambdas are represented similarly to functions in environment diagrams, but since they lack intrinsic names, the lambda symbol (λ) is used instead.

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.

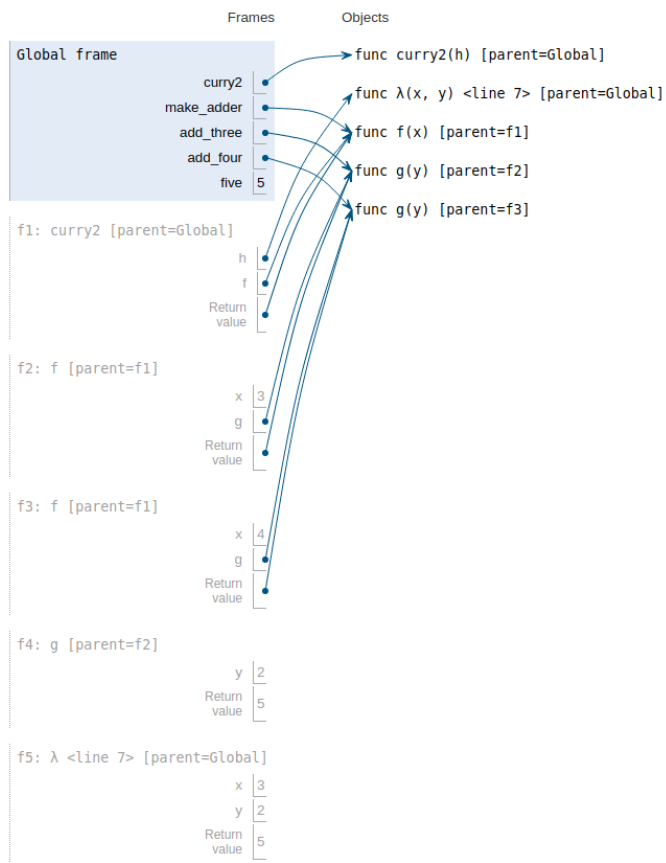
Questions

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

```

1 def curry2(h):
2     def f(x):
3         def g(y):
4             return h(x, y)
5         return g
6     return f
7 make_adder = curry2(lambda x, y: x + y)
8 add_three = make_adder(3)
9 add_four = make_adder(4)
10 five = add_three(2)

```



Solution: [pythontutor](#)

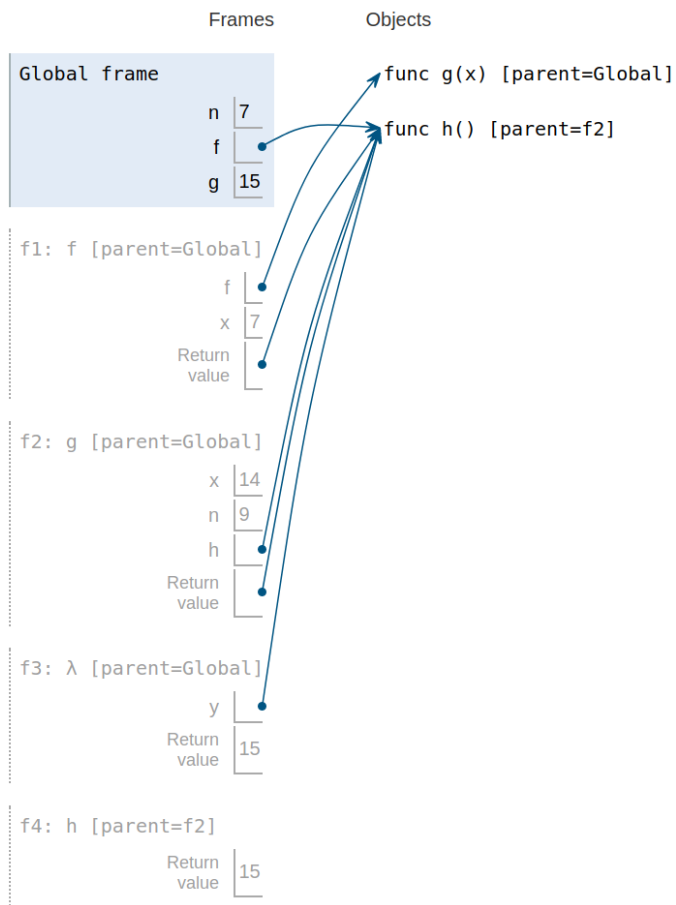
4 Higher Order Functions

1.2 Write `curry2` as a lambda function.

```
curry2 = lambda h: lambda x: lambda y: h(x, y)
```

1.3 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)
```



Video Walkthrough
[pythontutor](#)

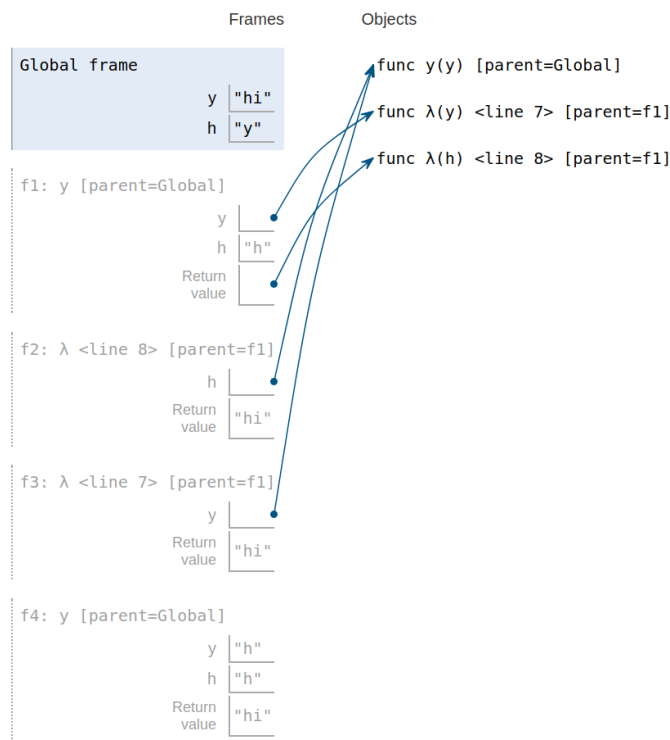
6 Higher Order Functions

- 1.4 The following question is more challenging than the previous ones. Nonetheless, it's a fun problem to try.

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

Note that 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"`

```
1 y = "y"
2 h = y
3 def y(y):
4     h = "h"
5     if y == h:
6         return y + "i"
7     y = lambda y: y(h)
8     return lambda h: y(h)
9 y = y(y)(y)
```



[Video walkthrough](#)
[pythontutor](#)

Writing Higher Order Functions

- 1.5 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`.

```
def keep_ints(cond, n):
    """Print out all integers 1..i..n where cond(i) is true

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

    i = 1
    while i <= n:
        if cond(i):
            print(i)
        i += 1
```

[Video walkthrough](#)

- 1.6 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`.

```
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
```

```
"""
```

```
def do_keep(cond):
```

```
    i = 1
```

```
    while i <= n:
```

```
        if cond(i):
```

```
            print(i)
```

```
        i += 1
```

```
    return do_keep
```

[Video Walkthrough](#)

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 to the right.

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

Note that 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.

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

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

Questions

- 1.7 Write a function `print_delayed` 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`.

```
def print_delayed(x):
    """Return a new function. This new function, when called,
    will print out x and return another function with the same
    behavior.

    >>> f = print_delayed(1)
    >>> f = f(2)
    1
    >>> f = f(3)
    2
    >>> f = f(4)(5)
    3
    4
    >>> f("hi")
    5
    <function print_delayed> # a function is returned
    """

    def delay_print(y):
        -----
        return -----
    return delay_print
```

```
def print_delayed(x):  
    def delay_print(y):  
        print(x)  
        return print_delayed(y)  
    return delay_print
```

- 1.8 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".

```
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 _____
```

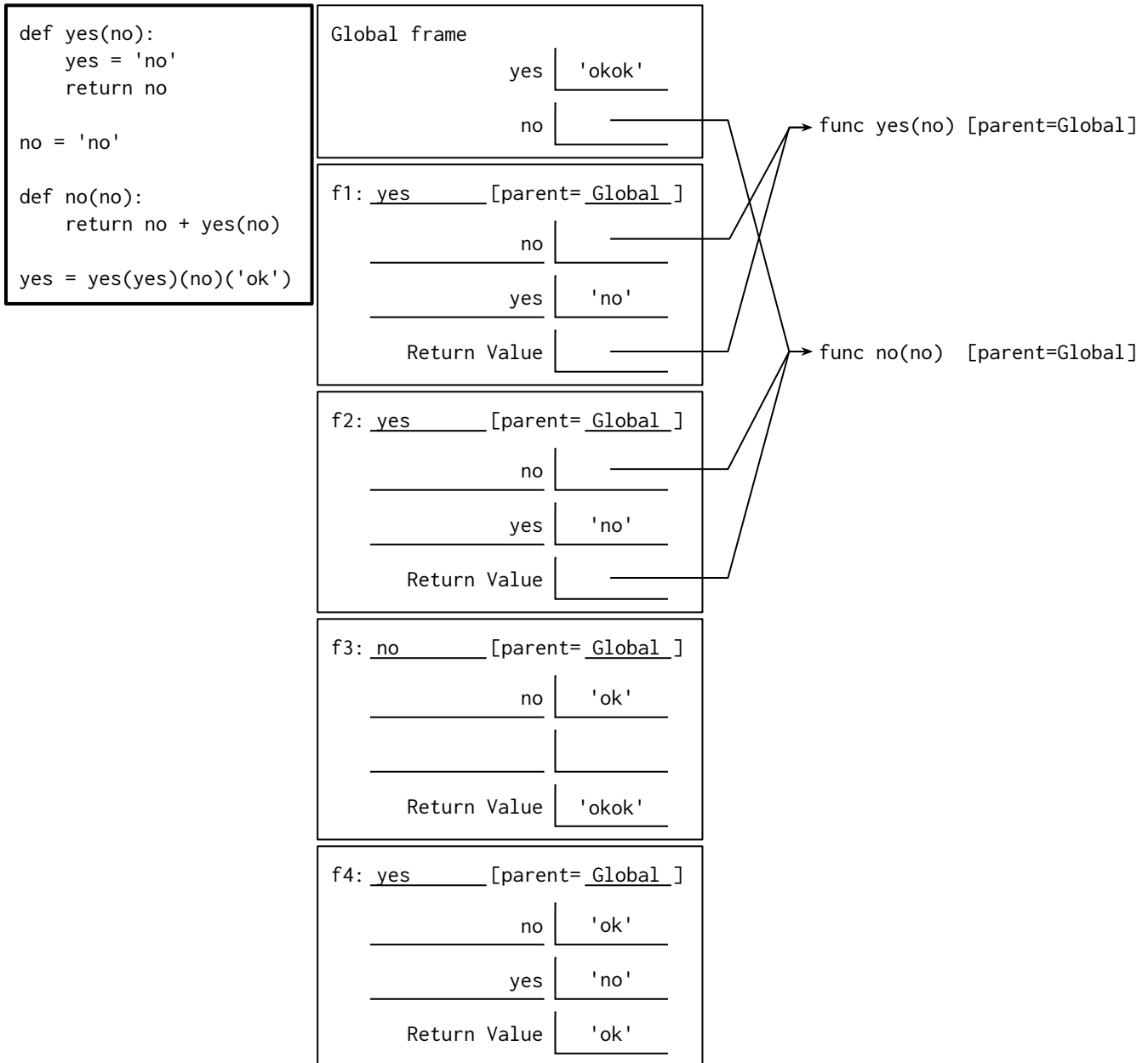
```
def print_n(n):
    def inner_print(x):
        if n <= 0:
            print("done")
        else:
            print(x)
        return print_n(n-1)
    return inner_print
```

1. Yes, No, but Sometimes Maybe?

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

You must list all bindings in the order they first appear in the frame.



Try stepping through the code slowly on [PythonTutor](#) if this diagram is inconsistent with what you believe to

be true. Some notes

When you have a higher order function and pass in multiple operands in one line, like the line that says `yes = yes(yes)(no)...` we do not evaluate all the operands at once. For example, we do not evaluate `no` until the call to `yes(yes)` has returned. This is because `yes(yes)` is tantamount to evaluating the operator for the next function call on `no`, and evaluating the operator always precedes evaluating the operand.

Draw pointers for non-primitive data, and write primitive data directly in the box. Primitive expressions in this course are almost always integers and strings, and non-primitive data are typically things like functions and lists (which we haven't learned about yet). A pointer represents a location in memory, so if you draw a pointer to a primitive that suggests both environment variables refer to the same data in memory when that may not be the case (i.e., suppose `x` in `f1` is 3 and so is `y` in `Global`. Modifying `x` should not affect the value of `y`, but if you draw a pointer to the same number that implies that a change to `x` will cause a change to `y`).

If there exists a variable in the global frame like `yes` and I set `yes` to be `"no"` in `f1`, that does *not* throw an error. Instead a new variable in `f1` called `yes` is created. How can that be? Didn't we learn that we can't change the value of variables existing in another frame? Python actually does not interpret `yes = "no"` in `f1` as an attempt to modify the global `yes`. The most common case in which you'd run into this sort of error would be if you wrote `x = x + 1` where `x` is *exclusively* defined in another frame. In this case, the program expects `x` to already exist (otherwise, how would we know the value of `x + 1`?), and if `x` only exists in global, then `x = x + 1` is interpreted as an attempt to change the existing value that is not in the current frame. Something like `x = 1` does *not* throw an error because there is no assumption that `x` already exists in some other frame.

Finally, recall that the value of an argument depends on the frame in which the function was called. This is because our order of evaluation is evaluate the operator, evaluate the operands, apply the function to the operands. Hence, the operands are evaluated before the function call is initiated, so they must be evaluated in the frame in which the call was performed. In this case, the majority of function calls are happening in the last line of code, which is in the global frame, so we're going to be evaluating `yes`'s and `no`'s in the global frame most of the time. When working with evaluation in places other than the global frame, we first look in the current frame for the value of an environment name. If we cannot find that environment name in the current frame, then we look in its parent and then the parent's parent and so on. The first time we encounter the name we are looking for, we stop looking up the frame hierarchy. In `f3` the name `no` exists, so when we try to evaluate `no` in `f3`, we run with the `f3` definition and stop looking through our hierarchy. Hence, `no` is `'ok'` as defined in `f3` instead of a function as defined in `global`.