Lecture #4: Higher-Order Functions

A Simple Recursion

- The Fibonacci sequence is defined

$$F_k = \begin{cases} k, & \text{for } k = 0, 1 \\ F_{k-2} + F_{k-1}, & \text{for } k > 1 \end{cases}$$

- ...which translates easily into Python:

```
def fib(n):
    """The Nth Fibonacci number, N>=0."""
    assert n >= 0
    if n <= 1:
        return n
    else:
        return fib(n-2) + fib(n-1)
```

- This definition works, but why is it so slow?

Redundant Calculation

- Consider the computation of fib(10).
- This calls fib(9) and fib(8), but then fib(9) calls fib(8) again and both fib(9) and the two calls to fib(8) call fib(7), so that fib(7) is called 3 times.
- Likewise, fib(6) is called 5 times, fib(7) is called 8 times, and so forth (in increasing Fibonacci sequence, interestingly enough.)
- Therefore, the time required (proportional to the number of calls) grows exponentially:
- As it turns out, fib(N) requires time roughly proportional to Φ^N, where the golden ratio $\Phi = (1 + \sqrt{5})/2$.

Avoiding Recalculation

- To compute the next Fibonacci number, we need the preceding two.
- Let's generalize and consider what it takes to compute N more:

```
def fib2(fk1, fk, k, n):
    """Assuming FK1 and FK2 are fib(K-1) and fib(K)
    in the Fibonacci sequence and that N>=K, return fib(N)."""
    if n == k:
        return fk
    else:
        return ________________________
def fib(n):
    if n <= 1:
       return n
    else:
       return fib2(0, 1, 1, n)
```

Avoiding Recalculation

- To compute the next Fibonacci number, we need the preceding two.
- Let's generalize and consider what it takes to compute N more:

```
def fib2(fk1, fk, k, n):
    """Assuming FK1 and FK2 are fib(K-1) and fib(K)
    in the Fibonacci sequence and that N>=K, return fib(N)."""
    if n == k:
        return fk
    else:
        return fib2(fk, fk1+fk, k+1, n)
def fib(n):
    if n <= 1:
       return n
    else:
       return fib2(0, 1, 1, n)
```

Tail Recursion and Repetition

- In this last version, whenever fib2 is called recursively, the value of that call is immediately returned.
- This property is called *tail recursion.*

```
def fib2(fk1, fk, k, n):
    if n == k: return fk
    else:      return fib2(fk, fk1+fk, k+1, n)
def fib(n):
    if n <= 1: return n
    else:      return fib2(0, 1, 1, n)
```

- It is this sort of process that is easily expressed as a repetition.
- Parameters become variables; initial call on fib2 inside fib initializes them; each tail-recursive call updates them. Iterative equivalent:

```
def fib3(n):
    if n <= 1: return n
    fk1, fk, k = 0, 1, 1
    while n != k:
        fk1, fk, k = fk, fk1+fk, k+1
    return fk
```

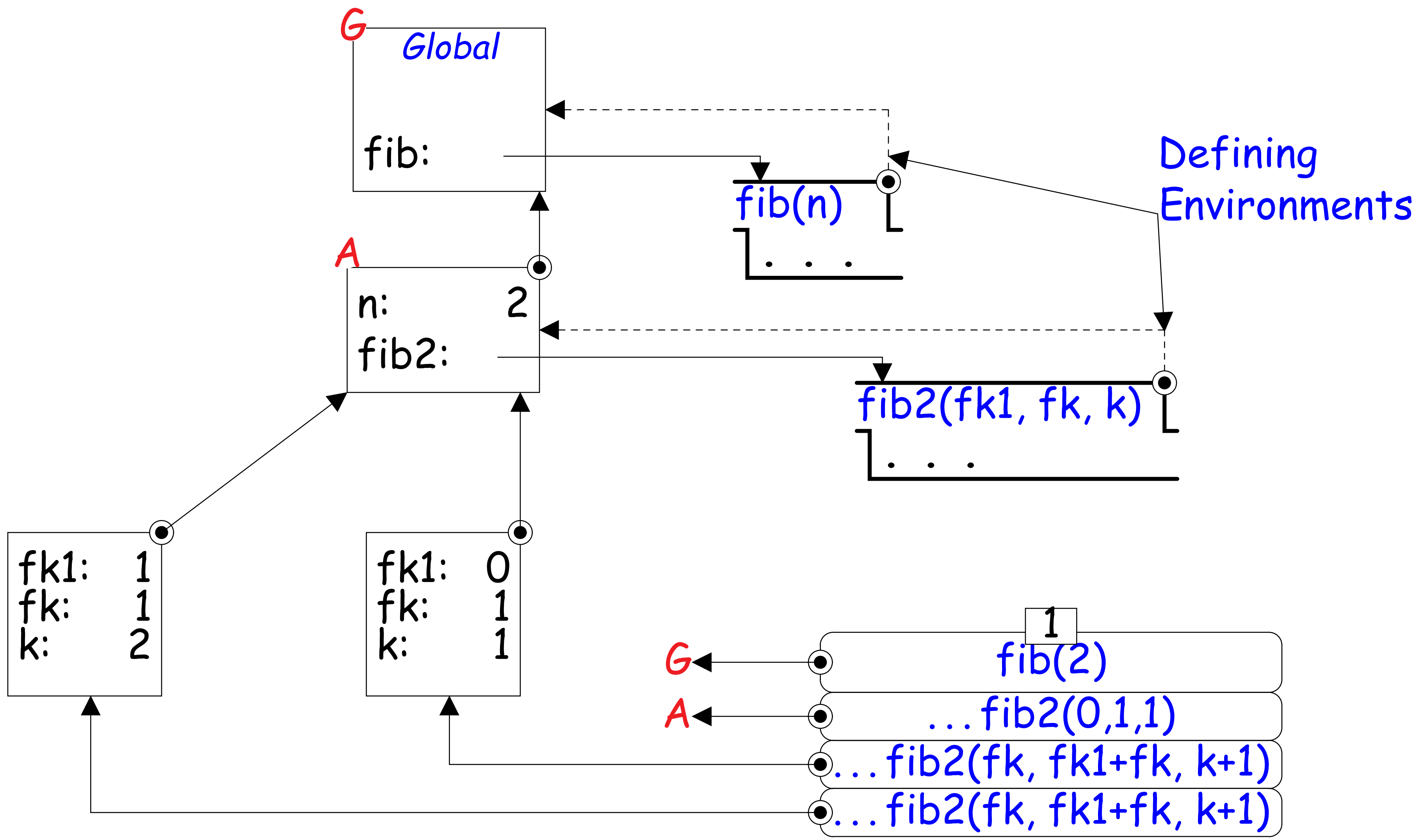
Nested Functions

- In the last recursive version, fib2 function is an auxiliary function, used only by fib.
- It makes sense to tuck it away inside fib, like this:

```
def fib(n):
    def fib2(fk1, fk, k):
        if n == k: return fk
        else:      return fib2( fk, fk1+fk, k+1)

    if n <= 1: return n
    else:      return fib2(0, 1, 1)
```

- I've taken the liberty here of removing the parameter n from fib2: it's always the same as the outer n and never changes.
- (See it here).

Nested Functions and Environments



Defining Environments

- Each function value is attached to the environment frame in which the **def** statement that created it was evaluated.
- Since the **def** for `fib` was evaluated in the global frame, the resulting function value bound to `fib` is attached to the global frame.
- Since the **def** for `fib2` was evaluated in the local frame of an execution of `fib`, the resulting function value is attached to that local frame.
- When a user-defined function value is called, the local frame that is created for that call is linked to the defining frame of the function.

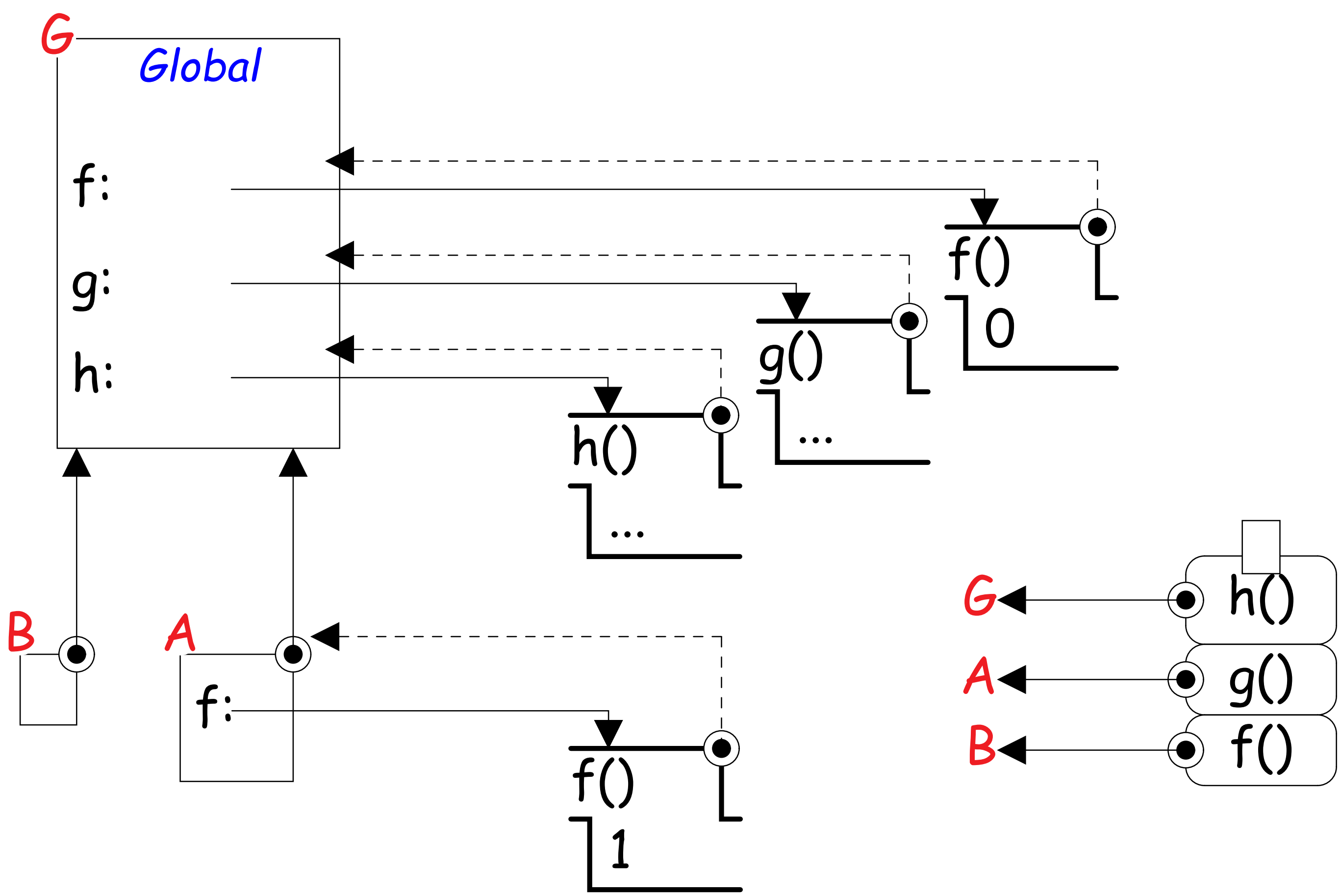
Do You Understand the Machinery? (I)

What is printed (0, 1, or error) and why?

```
def f():
    return 0

def g():
    print(f())

def h():
    def f():
        return 1
    g()

h()
```

Answer (I)

The program prints `0`. At the point that `f` is called, we are in the situation shown below:



So we evaluate f in an environment (B) where it is bound to a function that returns 0.

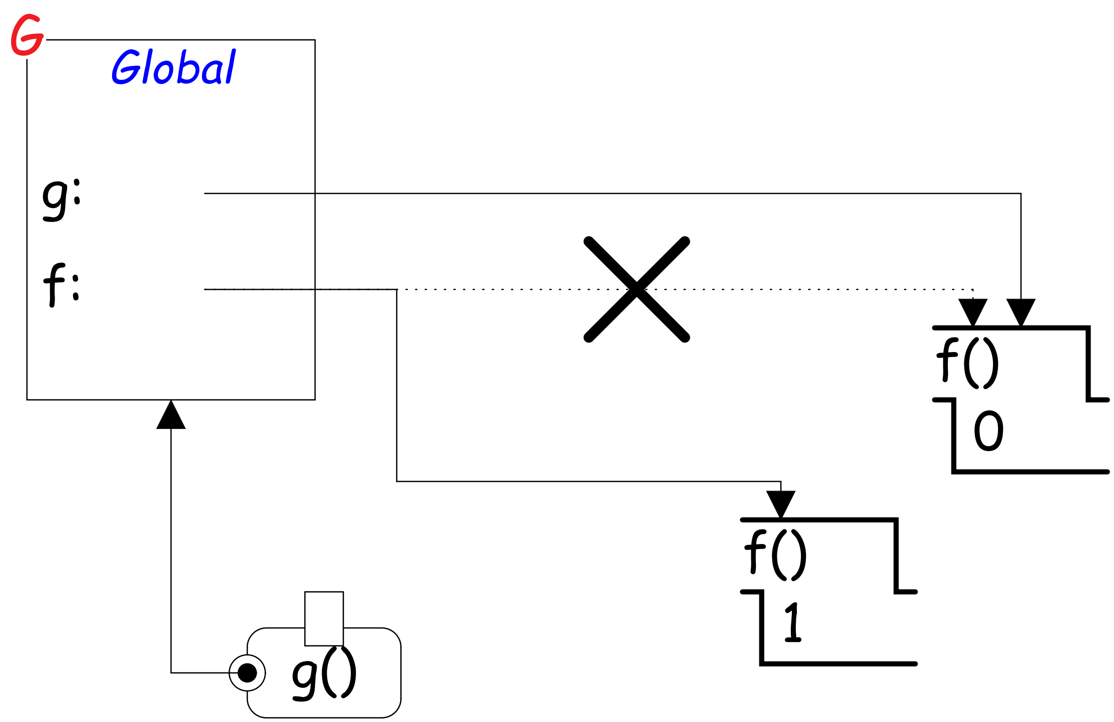
Do You Understand the Machinery? (II)

What is printed (0, 1, or error) and why?

```
def f():
    return 0

g = f

def f():
    return 1

print(g())
```

Answer (II)

The program prints 0 again:



At the time we evaluate f to assign it to g, it has the value indicated by the crossed-out dotted line, so that is the value g gets. The fact that we change `f`'s value later is irrelevant, just as `x = 3; y = x; x = 4; print(y)` prints 3 even though x changes: y doesn't remember where its value came from.

Do You Understand the Machinery? (III)

What is printed (0, 1, or error) and why?

```
def f():
    return 0

def g():
    print(f())

def f():
    return 1

g()
```

Answer (III)

This time, the program prints `1`. When g is executed, it evaluates the name 'f'. At the time that happens, f's value has been changed (by the third **def**), and that new value is therefore the one the program uses.

Functions As Templates

- If we think of a function body as a template for a computation, parameters are "blanks" in that template.
- For example:

```
def sum_squares(N):
    k, sum = 0, 0
    while k <= N:
        sum, k = sum+k**2, k+1
    return sum
```

is a template for an infinite set of computations that add squares of numbers up to 0, 1, 2, 3, . . . , in place of the N.

Functions on Functions

- Likewise, function parameters allow us to have templates with slots for *computations*:

```
def summation(N, f):
    k, sum = 1, 0
    while k <= N:
        sum, k = sum+f(k), k+1
    return sum
```

- Generalizes **sum_squares**. We can write `sum_squares(5)` as:

```
def square(x):
    return x*x
summation(5, square)
```

- or (if we don't really need a "square" function elsewhere), we can create the function argument anonymously on the fly:

```
summation(5, lambda x: x*x)
```

Lambda

- In Python, **lambda** is just an abbreviation.
- Writing lambda *PARAMS*: *EXPRESSION* is the same as writing NAME, where NAME is a name that appears nowhere else in the program and is defined by

```
def NAME(PARAMS):
    return EXPRESSION
```

  evaluated in the same environment in which the original **lambda** was.
- Now we can write any number of summations succinctly:

```
summation(10, lambda x: x**3)       # Sum of cubes
summation(10, lambda x: 1 / x)      # Harmonic series
summation(10, lambda k: x**(k-1) / factorial(k-1))
                                    # Approximate e**x
```

Functions that Produce Functions

- Functions are *first-class values,* meaning that we can assign them to variables, pass them to functions, and return them from functions.
- Example, let's generalize the class of functions like

```
def h(x):  return abs(x) + (-x)
```

- So that we can produce functions that add any two functions:

```
def add_func(f, g):
    """Return function that returns F(x)+G(x) for argument x."""
    def adder(x):              #
        return f(x) + g(x)     # or return lambda x: f(x) + g(x)
    return adder               #

from operator import abs, neg   # neg is unary -
h = add_func(abs, neg)
>>> print(h(-5))
10
```

Generalize!

- Let's make a general function-combining function (that goes beyond addition):

```
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x), g(x))
        return val
    return combined
```

- Now `add_func` is just an application:

```
from operator import add, neg
add_func = __________________
```

- What do the environments look like here? Think about it and try it out.

Generalize!

- Let's make a general function-combining function (that goes beyond addition):

```
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x), g(x))
        return val
    return combined
```

- Now `add_func` is just an application:

```
from operator import add, neg
add_func = combine_funcs(add)
```

- What do the environments look like here? Think about it and try it out.