Lecture #5: Higher-Order Functions

Announcements:

- Make sure that you have registered electronically with *our* system (not just TeleBEARS).
- Attend a discussion/lab in which you can fit; don't worry about TeleBEARS lab/discussion time once it allows you to register.
- Concurrent enrollment students should all get in (once fees are paid, that is).

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 FK F[K-1] and F[K] in the Fibonacci
    sequence numbers and N>=K, return F[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 an *iteration.*

Explicit Iteration

- In the Python, C, Java, and Fortran communities, it is more usual to be explicit about repetition, rather than using tail recursion.
- The simplest form is **while**

`while` *Condition*:
    Statements

means "If condition evaluates to a true value, execute statements and repeat the entire process. Otherwise, do nothing."

Explicit Iteration in fib

- Original version, again:

```
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)
```

- As an explicit iteration:

```
def fib(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
```

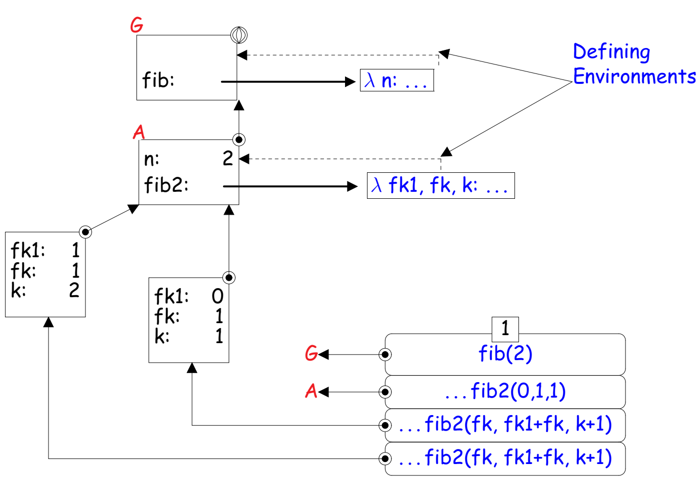

Nested Functions

- 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.
- But to explain how this works, we'll have to extend the environment model just a bit.

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 attached to the defining frame of the function.

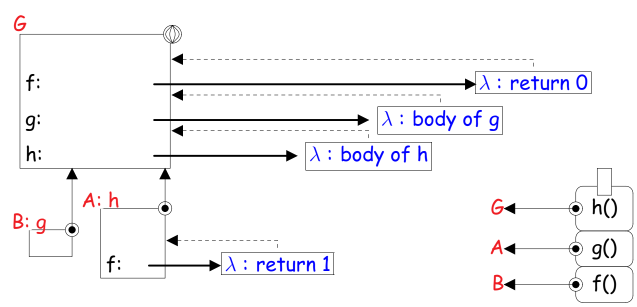
Do You Understand the Machinery? (I)

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. (B: g means that frame B was created to execute a call to g).

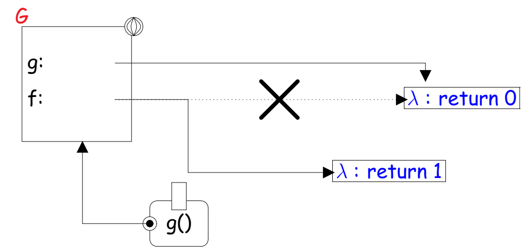
Do You Understand the Machinery? (II)

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 evan 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)
```

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:

```
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             #

h = add_func(abs, lambda x: -x)
>>> print(h(-5))
10
```

- Generalize the example:

```
def combine_funcs(op, f, g):
    return lambda x:  op(f(x), g(x))
# Now add_func = lambda f, g:  combine_funcs(sum, f, g)
```

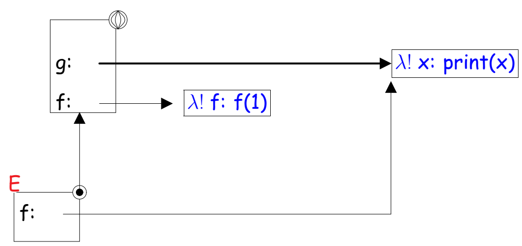

Do You Understand the Machinery? (IV)

What is printed: (1, infinite loop, or error) and why?

```
def g(x):
    print(x)

def f(f):
    f(1)

f(g)
```


Answer (IV)

This prints 1. When we reach f(1) inside f, the call expression, and therefore the name f, evaluated in the environment E, where the value of f is the global function bound to g:



An Aside: Notations

- To introduce environments, I used arrows to indicate connections between boxes to show these relationships graphically.
- But for serious use, that notation gets cluttered rapidly.
- Also, the Python Tutor software does not use it, favoring textual labels instead.
- There is a link to our official rules for building environment diagrams with the Python tutor notation on the class web page:

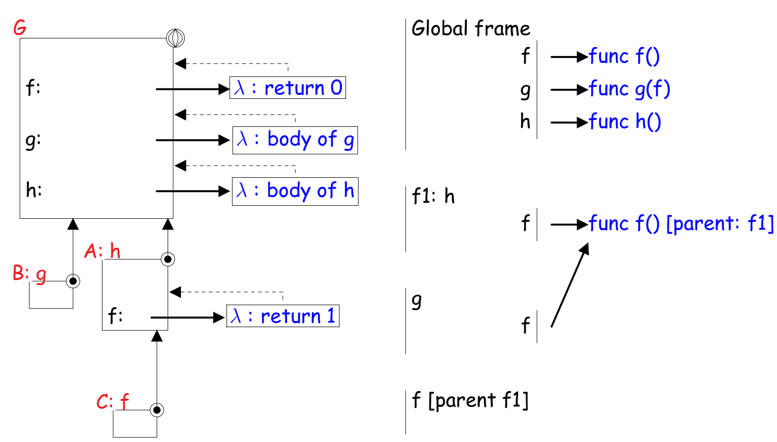
  https://inst.eecs.berkeley.edu/ cs61a/sp14/pdfs/environment-diagrams.pdf
- For these lectures, I'll generally use the more explicit style (with arrows linking frames) to show everything graphically, but we'll use the more compact Python tutor style for homework and tests.

Example of the Notations

Consider the following program, which will print 1:

```
def f():
   return 0

def g(f):
   print(f())

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

h()
```


Example, contd: Lecture notation and Python Tutor notation



Do You Understand the Machinery? (V)

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

```
def f():
    return 0

def g():
    return f()

def h(k):
    def f():
        return 1
    p = k
    return p()

print(h(g))
```


Answer (V)

This prints 0. Function values are attached to current environments when they are first created (by lambda or def). Assignments (such as to p) don't themselves create new values, but only copy old ones, so that when p is evaluated, it is equal to k, which is equal to g, which is attached to the global environment.

Observation: Environments Reflect Nesting

- From what we've seen so far:

  Linking of environment frames $\Longleftrightarrow$ *Nesting of definitions.*

- For example, given

```
def f(x):
    def g(x):
        def h(x):
            print(x)
        ...
    ...
```

  The structure of the program tells you that the environment in which *print(x)* is evaluated will always be a chain of 4 frames:

  Frame for *h* $\Longrightarrow$ Frame for *g* $\Longrightarrow$ Frame for *f* $\Longrightarrow$ Global frame.

- However, when there are multiple local frames for a particular function lying around, environment diagrams can help sort them out.

Do You Understand the Machinery? (VI)

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

```
def f(p, k):
    def g():
        print(k)
    if k == 0:
        f(g, 1)
    else:
        p()
f(None, 0)
```


Answer (VI)

This prints 0. There are two local frames for f when p() is called. In the first one, k is 0; in the second, it is 1. When p() is called, its value comes from the value of g that was created *in the first frame*, where k is 0.