Lecture #5: Exercising Environments

Announcements:

• Discussion orientation attendance is a bit low. Tutorials aren’t intended to present reviews of material, and they assume that you have attended orientation.
• As of Thursday, CS10 had additional seats. If you find you are not ready for CS61A, consider switching to CS10.
• Please see Piazza message @318 for test times and for the form requesting alternative times in the case of time conflicts.
• Ask questions on the Piazza thread for today’s lecture (@346).

Today

• In this lecture, there is nothing new!
• We’ll just look at illustrations of the rules set down previously.

Example I: Which Definition?

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

```python
def f():
    return 0

def g():
    print(f())

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

[Python Tutor]

Answer I

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

```
          Global frame
          ↑             ↑
f  func f()  [↑ Global]
g  func g()  [↑ Global]
h  func h()  [↑ Global]
```

```
def f():
    return 0
def g():
    print(f())
def h():
def f():
    return 1
g()
h()
```

```
          Global frame
          ↑             ↑
f  func f()  [↑ Global]
g  func g()  [↑ Global]
h  func h()  [↑ Global]
```

So we evaluate `f` in an environment (f2) where it is bound to a function that returns 0.
Example II: Redefinition after Assignment

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

```python
def f():
    return 0

g = f

def f():
    return 1

print(g())
```

[Python Tutor]

Answer II

The program prints 0 again:

```python
def f():
    return 0
g = f
def f():
    return 1
print(g())
```

At the time we evaluate `f` in the assignment 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.

Example III: Redefinition

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

```python
def f():
    return 0

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

g()```

[Python Tutor]

Answer III

This time, the program prints 1. When `g` is executed, it evaluates the name `f`’s. 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.
Example IV: Which Definition?
What is printed: (1, infinite loop, or error) and why?
```
def f(f):
    f(1)

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

[Python Tutor]

Answer IV
This prints 1. When we reach \( f(1) \) inside \( f \), the call expression, and therefore the name \( f \), is evaluated in the environment starting at frame \( f1 \), where the value of \( f \) is the global function bound to \( g \):

![Python Tutor Diagram](image)

Example V: Which Definition?
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))
```

[Python Tutor]

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 $\iff$ Nesting of definitions.
- For example, given:
  ```python
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:
  - A local frame for `h` linked to ...
  - A local frame for `g` linked to ...
  - A local frame for `f` linked to ...
  - The global frame.
- However, when there are multiple local frames for a particular function lying around, environment diagrams can help sort them out.

Example VI: Multiple Executions of Def

What is printed: (0, 1, or `error`) and why?
```python
def f(p, k):
def g():
    print(k)
if k == 0:
    f(g, 1)
else:
p()
f(None, 0)
```

Example VII: Assign to Parameter

What is printed (4 2, 5 3, or 4 3) and why?
```python
def f(x):
x = x + 1
y = 4
f(y)
x = 2
f(x)
print(y, x)
```
Answer VII

The program prints "4 2". During the execution of f, the formal parameter x resides in a new local frame. Anything done to it has no effect on any variables in other frames, such as in the global frame from which f is called.

Example VIII: Assign to Outer Parameter?

What is printed (3, 4, or error) and why?

```python
def f(x):
def g(y):
x = y
g(4)
return x
print(f(3))
```

Answer VIII

In the call to g, the assignment to x creates a new binding of x in the local frame created by the call to g. It is unrelated to the parameter of f, which is bound in a different local frame. Hence, the call to g has no effect and the argument to f is returned unchanged.

Example IX: Delayed Recursion

What does this print, and why?

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

print_sums(1)(3)(5)
```

[Python Tutor]
**Answer IX**

The call

```
print_sums(1)(3)(5)
```

produces the same result as

```
g1 = print_sums(1)
g2 = g1(3)
g2(5)
```

A call `print_sums(x)` returns a function that

- Prints `x` as a side-effect, and
- Returns a function that, when called with argument `y`, will do exactly the same thing, but with `x+y` instead of `x`.

So these calls will

- First print 1 and return `g1`,
- which when called with 3, will print 4 (= 1+3) and return `g2`,
- which when called with 5, will print 9 (= 4+5), and return...

---

**Example X: Currying**

- The term *currying* refers to converting a multi-argument function into one that takes one argument and returns a function that takes the next argument, and so on, until it finally produces the original function's result after consuming the last argument.
- The name comes from Haskell Curry, who did not invent it.
- In fact, to name it after its inventor, we'd have to say "Frege-ing" or perhaps "Schönfinkeling".
- We could define the process for two arguments like this:

  ```python
def curry2(f):
    return lambda x: lambda y: f(x, y)

from operator import add
print(curry2(add)(30)(12)) # Prints a function value
print(curry2(add)(30))
```

[Python Tutor]