Environment Diagrams and HOFs

Computer Science Mentors CS 88

February 8 - 13

1 Environment Diagrams

- Creating a Function
  1. Draw the func <name>(<arg1>, <arg2>, ...)
  2. The parent of the function is wherever the function was defined (the frame we're currently in, since we're creating the function).
  3. If we used def, make a binding of the name to the value in the current frame.
- Calling User Defined Functions
  1. Evaluate the operator and operands.
  2. Create a new frame; the parent is whatever the operator's parent is. Now this is the current frame.
  3. Bind the formal parameters to the argument values (the evaluated operands).
  4. Evaluate the body of the operator in the context of this new frame.
  5. After evaluating the body, go back to the frame that called the function.
- Assignment
  1. Evaluate the expression to the right of the assignment operator (=).
  2. Bind the variable name to the value of the expression in the identified frame. Be sure you override the variable name if it had a previous binding.
- Lookup
  1. Start at the current frame. Is the variable in this frame? If yes, that's the answer.
  2. If it isn't, go to the parent frame and repeat 1.
  3. If you run out of frames (reach the Global frame and it's not there), complain.

1. Draw the environment diagram for evaluating the following code

```
def dessef(a, b):
    c = a + b
    b = b + 1
b = 6
dessef(b, 4)
```

2. Draw the environment diagram for evaluating the following code

```
def foo(x, y):
    foo = bar
    return foo(bar(x, x), y)

def bar(z, x):
    return z + y

y = 5
foo(1, 2)
```

2 Higher Order Functions

A **higher order function** (HOF) is a function that does at least one of the following:

- *accepts* at least one function as an argument
- *returns* a function

HOFs utilize the concept of treating **functions** as *data* just like any type of value such as integers, strings, lists, booleans, etc.

Functions as Arguments

Taking in functions as arguments can help generalize our code. Imagine we have a function `mul-by-2` which will take in a list and multiply each element by 2. If we'd want to be able to do something similar to `mul-by-2` but apply a different operation, we'd have to make a different function, but nearly all the code between the two would be the same!

A way that generalizes this is a function that takes in two arguments, the list and a one argument function that will perform the operation we'd like. This function is known as `map`. Below is an example of applying a `cook` function to a list of various food items:

```
>>> map(cook, ["cow", "potato", "chicken", "corn"])
["burger", "fries", "fried chicken", "popcorn"]
```

Functions as Return Values

Often, we will need to write a function that returns another function. One way to do this is to define a function `inner` inside of a function `outer`, and `outer` will return the function `inner`.

Some cases where we might do this is:

- need additional information (in the example below, we needed information of the `name` of whom to greet)
- might need to track other variables that aren't included

```
def maker-greeter(greeting):
    def greet(name):
        print(greeting, name)
    return greet

>>> hello-greeter = make-greeter("Hello")
>>> greet("Alina")
Hello Alina
```

1. Implement `make_skipper`, which takes in a number `n` and outputs a function. When this function takes in a number `x`, it prints out all the numbers between `0` and `x` inclusive, skipping every `n`th number.

```
def make_skipper(n):
    """
    >>> a = make_skipper(2)
    >>> a(5)
    1
    3
    5
    """
```

2. Implement `apply_func` which takes in a one argument function `f` and returns a one argument function. The returned function takes in a list `lst` and applies `f` to each element in `lst`.

```
def apply_func(f):
    """
    >>> g = apply_func(abs)
    >>> lst = [1, -1, 2, -2]
    >>> g(lst)
    >>> lst
    [1, 1, 2, 2]
    """
```

3 WWPD

1. What does the bottom function call return?

```
>>> apple = 4
>>> def orange(apple):
...   apple = 5
...   def plum(x):
...     return x * 2
...   return plum
...
>>> orange(apple)('hiii')
```

2. What is returned in line 1 and line 2? (Recommended: Draw an environment diagram!)

```
>>> def f(g, f):
...   return g(f)
...
>>> def foo(g, h):
...   return h * g(h)
...
>>> def h(i):
...   return 5
...
>>> f(h, foo)

>>> f(h, foo(h, 3))
```

4 Optional Challenging Problems!

1. Draw the environment diagram for evaluating the following code

```
def spain(japan, iran):
    def world(cup, egypt):
            return japan-poland
    return iran(world(iran, poland))

def saudi(arabia):
    return japan + 3

japan, poland = 3, 7
spain(poland+1, saudi)
```

2. Implement the function `filter_out` that takes in a list `lst` and returns a one argument function, let's arbitrarily call this `g`. `g` takes in a one argument function `f` and returns a pair — a new list containing only the elements of `lst` that return True when passed in to `f`, and a one argument function that behaves identically to `g` but operates on the filtered list.

```
def filter_out(lst):
    """
    >>> #Here are a couple of helper functions
    >>> def less_than_4(x):
    >>>     return x < 4
    >>> def divisible_by_2(x):
    >>>     return x % 2 == 0
    >>> g = filter_out([1, 2, 3, 4, 5])
    >>> lst, b = g(less_than_4)
    >>> lst
    [1, 2, 3]
    >>> lst, c = b(divisible_by_2)
    >>> lst
    [2]
    """
```