

# ENVIRONMENT DIAGRAMS AND RECURSION 2

COMPUTER SCIENCE 61A

February 4, 2016

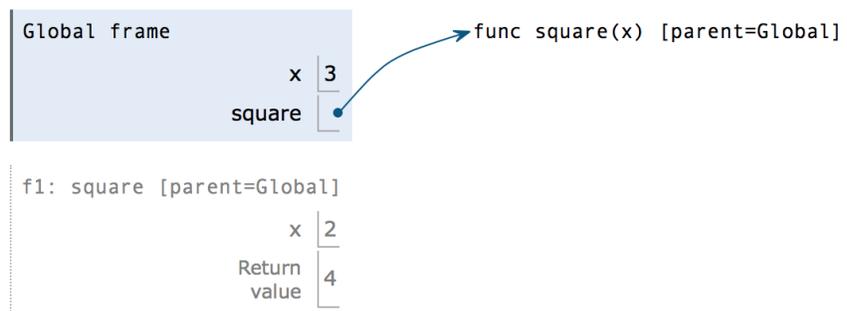
## 1 Environment Diagrams

An **environment diagram** keeps track of all the variables that have been defined and the values they are bound to.

```
x = 3
```

```
def square(x):  
    return x ** 2
```

```
square(2)
```



When you execute *assignment statements* in an environment diagram (like `x = 3`), you need to record the variable name and the value:

1. Evaluate the expression on the right side of the `=` sign
2. Write the variable name and the expression's value in the current frame.

When you execute *def statements*, you need to record the function name and bind the function object to the name.

1. Write the function name (`square`) in the frame and point it to a function object (`func square(x) [parent=Global]`). The `[parent=Global]` denotes the frame in which the function was *defined*.

When you execute a *call expression* (like `square(2)`), you need to create a new frame to keep track of local variables.

1. Draw a new frame.<sup>a</sup> Label it with
  - a unique index (`f1`, `f2`, `f3` and so on)
  - the **intrinsic name** of the function (`square`), which is the name of the function object itself. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.
  - the parent frame (`[parent=Global]`)
2. Bind the formal parameters to the arguments passed in (e.g. bind `x` to `3`).
3. Evaluate the body of the function.

If a function does not have a return value, it implicitly returns `None`. Thus, the “Return value” box should contain `None`.

---

<sup>a</sup>Since we do not know how built-in functions like `add(...)` or `min(...)` are implemented, we do *not* draw a new frame when we call built-in functions.

---

## 1.1 Environment Diagram Questions

---

1. Draw the environment diagram so we can visualize exactly how Python evaluates the code. What is the output of running this code in the interpreter?

```
>>> from operator import add
>>> def sub(a, b):
...     sub = add
...     return a - b
>>> add = sub
>>> sub = min
>>> print(add(2, sub(2, 3)))
```

2. Draw the environment diagram for the following code:

```
from operator import add
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f
```

```
make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

---

## 2 Recursion

---

A *recursive* function is a function that calls itself. Here's a recursive function:

```
def factorial(n):  
    if n == 0 or n == 1:  
        return 1  
    else:  
        return n * factorial(n-1)
```

Although we haven't finished defining `factorial`, we are still able to call it since the function body is not evaluated until the function is called. We do have one *base case*: when `n` is 0 or 1. Now we can compute `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` in terms of `factorial(2)`, and `factorial(4)` – well, you get the idea.

There are *three* common steps in a recursive definition:

1. *Figure out your base case*: What is the simplest argument we could possibly get? For example, `factorial(0)` is 1 by definition.
2. *Make a recursive call with a simpler argument*: Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the “leap of faith”. For `factorial`, we reduce the problem by calling `factorial(n-1)`.
3. *Use your recursive call to solve the full problem*: Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For `factorial`, we just multiply  $(n - 1)!$  by  $n$ .

### 2.1 Cool recursion questions!

---

1. Create a recursive countdown function that takes in an integer `n` and prints out a countdown from `n` to 1. The function is defined on the next page.

First, think about a base case for the `countdown` function. What is the simplest input the problem could be given?

After you've thought of a base case, think about a recursive call with a smaller argument that approaches the base case. What happens if you call `countdown(n - 1)`?

Then, put the base case and the recursive call together, and think about where a print statement would be needed.

```
def countdown(n):  
    """  
    >>> countdown(3)  
    3  
    2  
    1  
    """
```

2. Is there an easy way to change `countdown` to count up instead?

3. Write a recursive function that sums the digits of a number `n`. Assume `n` is positive. You might find the operators `//` and `%` useful.

```
def sum_digits(n):  
    """  
    >>> sum_digits(7)  
    7  
    >>> sum_digits(30)  
    3  
    >>> sum_digits(228)  
    12  
    """
```

### 3 Iteration vs. Recursion

---

We've written factorial recursively. Let's compare the iterative and recursive versions:

```
def factorial_recursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n-1)
```

```
def factorial_iterative(n):
    total = 1
    while n > 1:
        total = total * n
        n = n - 1
    return total
```

Let's also compare fibonacci.

```
def fib_recursive(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)
```

```
def fib_iterative(n):
    current, next = 0, 1
    while n > 0:
        current, next = next, current + next
        n = n - 1
    return current
```

For the recursive version, we copied the definition of the Fibonacci sequence straight into code! The  $n$ th Fibonacci number is simply the sum of the two before it. In iteration, you need to keep track of more numbers and have a better understanding of the code.

Some code is easier to write iteratively and some recursively. Have fun experimenting with both!

---

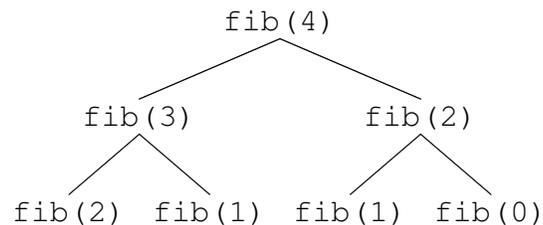
## 4 Tree Recursion

---

Consider a function that requires more than one recursive call. A simple example is the previous function:

```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called *tree recursion*, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

1. I want to go up a flight of stairs that has  $n$  steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me. Assume  $n$  is positive.

Before we start, what's the base case for this question? What is the simplest input?

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

Use those two recursive calls to write the recursive case:

```
def count_stair_ways(n):
```

2. The TAs want to print handouts for their students. However, for some unfathomable reason, both the printers are broken; the first printer only prints multiples of  $n_1$ , and the second printer only prints multiples of  $n_2$ . Help the TAs figure out whether or not it is possible to print an exact number of handouts!

First try to solve without a helper function. Also try to solve using a helper function and adding up to the sum.

```
def has_sum(total, n1, n2):  
    """  
    >>> has_sum(1, 3, 5)  
    False  
    >>> has_sum(5, 3, 5) # 0(3) + 1(5) = 5  
    True  
    >>> has_sum(11, 3, 5) # 2(3) + 1(5) = 11  
    True  
    """
```

---

## 5 Environment Diagram Rules

---

### 1. Creating a Function

- Draw the func `<name>(<arg1>, <arg2>, ...)`
- The parent of the function is wherever the function was defined (the frame we're currently in, since we're creating the function).
- If we used `def`, make a binding of the name to the value in the current frame.

### 2. Calling User Defined Functions

- Evaluate the operator and operands.
- Create a new frame; the parent is whatever the operator's parent is. Now this is the current frame.
- Bind the formal parameters to the argument values (the evaluated operands).
- Evaluate the body of the operator in the context of this new frame.
- After evaluating the body, go back to the frame that called the function.

### 3. Assignment

- Evaluate the expression to the right of the assignment operator (`=`).
- If nonlocal, find the frame that has the variable you're looking for, starting in the parent frame and ending just before the global frame (via Lookup rules). Otherwise, use the current frame. Note: If there are multiple frames that have the same variable, pick the frame closest to the current frame.
- 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.

### 4. Lookup

- Start at the current frame. Is the variable in this frame? If yes, that's the answer.
- If it isn't, go to the parent frame and repeat 1.
- If you run out of frames (reach the Global frame and it's not there), complain.

### 5. Tips

- You can only bind names to values. No expressions (such as `3 + 4`) are allowed in environment diagrams!
- Frames and Functions both have parents.

---

## 6 Challenge! (Midterm Practice)

---

1. Draw the environment diagram for the following code: (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")

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```

2. Draw an environment diagram for the following code:

```
def rec(x, y):
    if y > 0:
        return x * rec(x, y - 1)
    return 1
rec(3, 2)
```

Bonus question: what does this function do?

Global frame	
.....	
.....	
.....	
.....	
.....	

f1: ..... [parent=.....]	
.....	
.....	
.....	
Return Value	

f2: ..... [parent=.....]	
.....	
.....	
.....	
Return Value	

f3: ..... [parent=.....]	
.....	
.....	
.....	
Return Value	