RECURSION: *n.* SEE RECURSION 3

COMPUTER SCIENCE 61A

June 30, 2015

1 Recursion

A *recursive* function is a function that calls itself. Below is a recursive `factorial` 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 your 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.

1.1 Cool recursion questions!

1. Print out a countdown using recursion.

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

   First, think about a base case. 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 approches 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.

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

3. Write a function `recursive_mul(m, n)` that multiplies two numbers `m` and `n`. Assume `m` and `n` are positive integers. Use recursion, not `mul` or `*`!

   Hint: `5*3 = 5 + 5*2 = 5 + 5 + 5*1`.

   For the base case, what is the simplest possible input for `recursive_mul`?

   For the recursive case, what does calling `multiply(m - 1, n)` do? What does calling `multiply(m, n - 1)` do? Which one do we want to use?

```
def multiply(m, n):
    """
    >>> multiply(5, 3)
    15
    """
```

4. Write a procedure `expt(base, power)`, which implements the exponent function. For example, `expt(3, 2)` returns 9, and `expt(2, 3)` returns 8. Assume `power` is always a non-negative integer. Use recursion, not `pow`!

```
def expt(base, power):
```

5. 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
    """
```

6. Below is the iterative version of `is_prime`, which returns `True` if positive integer `n` is a prime number and `False` otherwise:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

   Implement the recursive `is_prime` function. Do not use a while loop, use recursion.

```
def is_prime(n):
```

7. Write `sum_primes_up_to(n)`, which sums up every prime up to and including `n`. Assume you have an `is_prime(n)` predicate.

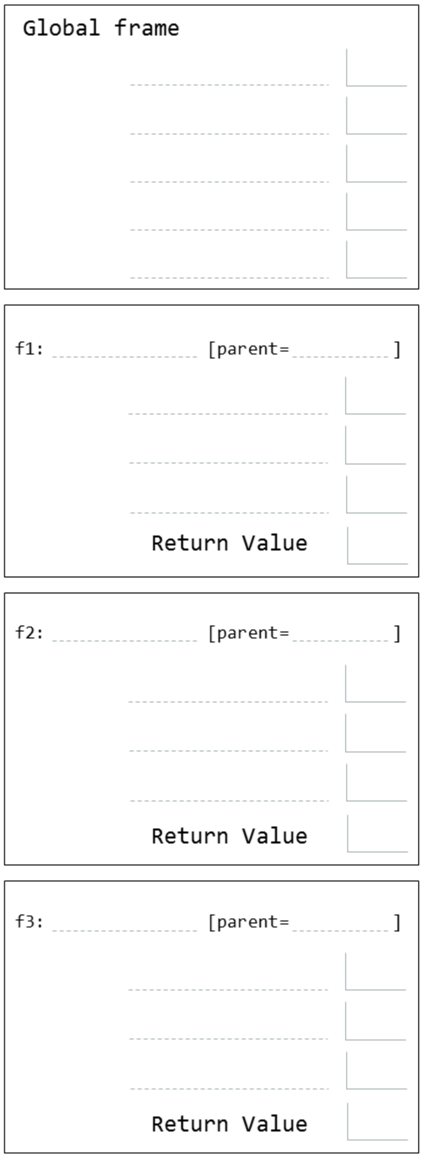
```
def sum_primes_up_to(n):
```

1.2 Recursive Environment Diagram!

1. 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?



2 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
```

Notice, while the recursive function "works" until n is less than or equal to 0, the iterative function "works" while n is greater than 0. They're essentially the same.

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 nth fibonacci number is simply the sum of the two before it. Iteratively, 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!