HIGHER ORDER FUNCTIONS 2

COMPUTER SCIENCE 61A

January 30, 2013

1 Warmup Questions

1. Here is one method to check if a number is prime:

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

How does this function work?

> **Solution:** It checks if the argument, a number `n`, is prime by checking if it is divisible by any number between 1 and itself.

This is a decent way of testing if a number is prime, but looping `k` all the way to `n` might be a bit cumbersome. As a little bonus question, can you think of a better place to stop?

> **Solution:** The square root of a number. If d divides n, then n/d also divides n. d and n/d cannot *both* be greater than $\sqrt{n}$.

Using the `is_prime` function, fill in the following function, which generates the n^{th} prime number. For example, the 2^{nd} prime number is 3, the 5^{th} prime number is 11, and so on.

```
def nth_prime(n):
```

Solution:

```
count, curr = 1, 2
while count < n:
    curr = curr + 1
    if is_prime(curr):
        count = count + 1
return curr
```

2. Now, what if we wanted to print a sequence of primes up to the n^{th} prime. What would be a simple way to do this?

Solution: Insert a `print` statement inside the `if`-statement, so that the prime numbers are printed as they are discovered.

3. The Fibonacci sequence is a famous sequence in mathematics where each term is generated by adding the two previous terms: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$ Using a `while` loop, write a function that would find the n^{th} Fibonacci number. For example, the 4^{th} number would be 2 and the 6^{th} number would be 5.

```
def nth_fibo(n):
```

Solution:

```
count, curr, next = 1, 0, 1
while count < n:
    curr, next = next, curr + next
    count += 1
return curr
```

2 Environment Diagrams

Environment diagrams will feature prominently in CS61A, so here is a simple one to try for practice. Environment diagrams can help you understand difficult coding problems, and also give you an idea of what's happening inside the interpreter.

Write the environment diagram for the following code:

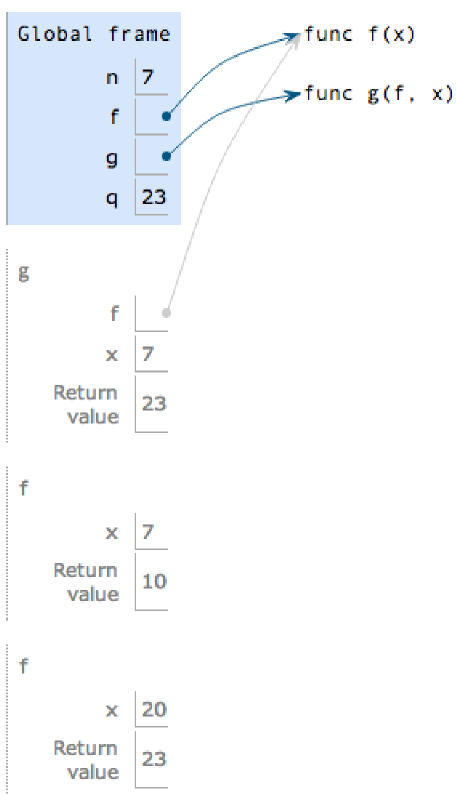
```
n = 7

def f(x):
    return x + 3

def g(f, x):
    return f(f(x)*2)

q = g(f, n)
```

Solution:



3 Functions

A function that manipulates other functions as data is called a *higher order function* (HOF). For instance, a HOF can be a function that takes functions as arguments, returns a function as its value, or both.

4 Functions as Argument Values

Suppose we would like to square or double every natural number from 1 to `n` and print the result as we go. Using the functions `square` and `double`, each of which are functions that take one argument that do as their name imply, fill out the following:

```
def square_every_number(n):
```

Solution:

```
    i = 1
    while i <= n:
        print(square(i))
        i += 1
```

```
def double_every_number(n):
```

Solution:

```
    i = 1
    while i <= n:
        print(double(i))
        i += 1
```

Note that the only thing different about `square_every_number` and `double_every_number` is just what function we call on `n` when we print it. Wouldn't it be nice to generalize functions of this form into something more convenient? When we pass in the number, couldn't we specify, also, what we want to do to each number $<$ n.

To do that, we can define a higher order function called `every`. `every` takes in the function you want to apply to each element as an argument, and applies it to n natural numbers starting from 1. So to write `square_every_number`, we can simply do:

```
def square_every_number(n):
    every(square, n)
```

Equivalently, to write `double_every_number`, we can write:

```
def double_every_number(n):
    every(double, n)
```

Note: These functions are not pure — as defined below, `every` will actually print values to the screen.

5 Questions

1. Now implement the function `every` that takes in a function `func` and a number `n`, and applies that function to the first n numbers from 1 and prints the result along the way:

   ```
   def every(func, n):
   ```

   Solution:

   ```
       i = 1
       while i <= n:
           print(func(i))
           i += 1
   ```

2. Similarly, implement the function `keep`, which takes in a function condition `cond` and a number `n`, and only prints a number from 1 to n to the screen if it fulfills the condition:

   ```
   def keep(cond, n):
   ```

   Solution:

   ```
       i = 1
       while i <= n:
           if cond(i):
               print(i)
           i += 1
   ```

6 Functions as Return Values

This problem comes up often: write a function that, given something, **returns a function** that does something else. The key message — conveniently emphasized — is that your function is supposed to return a function. For now, we can do so by defining an internal function within our function definition and then returning the internal function.

```
def my_wicked_function(blah):
    def my_wicked_helper(more_blah):
        ...
    return my_wicked_helper
```

That is the common form for such problems but we will learn another way to do this shortly.

7 Moar Questions

1. Write a function `and_add_one` that takes a function `f` as an argument (such that `f` is a function of one argument). It should return a function that takes one argument, and does the same thing as `f`, except adds one to the result.

   ```
   def and_add_one(f):
   ```

   Solution:

   ```
       def foo(x):
           return f(x) + 1
       return foo
   ```

2. Write a function `and_add` that takes a function `f` and a number `n` as arguments. It should return a function that takes one argument, and does the same thing as the function argument, except adds n to the result.

   ```
   def and_add(f, n):
   ```

   Solution:

   ```
       def foo(x):
           return f(x) + n
       return foo
   ```

3. The following code has been loaded into the python interpreter:

```
def skipped(f):
    def g():
        return f
    return g

def composed(f, g):
    def h(x):
        return f(g(x))
    return h

def added(f, g):
    def h(x):
        return f(x) + g(x)
    return h

def square(x):
        return x*x

def two(x):
    return 2
```

What will python output when the following lines are evaluated? Write "Error" if evaluating the line will result in an error.

```
>>> composed(square, two)(7)
```

Solution:

```
4
```

```
>>> skipped(added(square, two))()(3)
```

Solution:

```
11
```

```
>>> added(square, two)()(3)
```

Solution:

```
ERROR
```

```
>>> composed(two, square)(2)
```

Solution:

```
2
```

4. Python represents a programming community, and for things to run smoothly, there are some standards to keep things consistent. The following is the recommended style for documentation so that collaboration with other python programmers becomes standard and easy. If you're up to it, write your code at the very end, using `accumulate` as specified in next week's homework. We'll cover this more extensively later.:

```
def identity(x):
    return x

def lazy_accumulate(f, start, n, term):
    """
    Takes the same arguments as accumulate from next week's homework and
    returns a function that takes a second integer m and
    will return the result of accumulating the first n
    numbers starting at 1 using f and combining that with
    the next m integers.

    Arguments:
    f - the function for the first set of numbers.
    start - the value to combine with the first value in
            the sequence.
    n - the stopping point for the first set of numbers.
    term - function to be applied to each number before
           combining.

    Returns:
    A function (call it h) h(m) where m is the number of
    additional values to combine.

    >>> # The following does
    >>> # (1 + 2 + 3 + 4 + 5) + (6 + 7 + 8 + 9 + 10)
    >>> lazy_accumulate(add, 0, 5, identity)(5)
    55
    """
```

Solution:

```
    def second_accumulate(m):
        return accumulate(f, start, n + m, term)
    return second_accumulate
```