CONTROL AND HIGHER ORDER FUNCTIONS 2

COMPUTER SCIENCE 61A

September 1, 2013

1 Warmup Question

1. Draw the environment diagram for this code

```
n = 7

def f(x):
    return x + 3

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

m = g(f, n)
```

2 Control

Control refers to directing the computer to selectively choose which lines of code get executed

2.1 Conditional Statements

Conditional statements allow programs to execute different lines of code depending on the current state. A typical if-else set of statements will have the following structure:

```
if <conditional expression>:
    <suite of statements>
elif <conditional clause>:
    <suite of statements>
else:
    <suite of statements>
```

The else and elif statements are optional and you can have any number of elif statements. Here a conditional clause is something that evaluates to either `True` or `False`. The body of statements that get executed are the ones under the first true conditional clause. After a true conditional clause is found, the rest are skipped. Note that in python everything evaluates to `True` except `False`, `0`, `""`, and `None`. There are other things that evaluate to `False` but we haven't learned the yet.

```
>>> if 2+3:
...     print(6)
6
```

Here's some example code

```
>>> def mystery(x):
...     if x > 0:
...         print(x)
...     else:
...        x(mystery)
...
>>> mystery(5)
5
>>> mystery(-1)
TypeError: 'int' object is not callable
```

1. Write a simple function that takes in one input x, whose value is guaranteed to be between 0 and 100. if $x > 75$ then print "Q1". If $50 \leq x < 75$ then print "Q2". If $25 \leq xs < 50$ then print "Q3". If $x < 25$ then print "Q4".

```
def find_quartile(x):
```

2. Now try rewriting the function so that at most 4 lines of code inside the function will ever get executed.

```
def find_quartile(x):
```

2.2 Iteration

Using conditional statements we can ignore statements. On the other hand using iteration we can repeat statements multiple times. A common iterative block of code is the while statement. The structure is as follows:

```
while <conditional clause>:
    <body of statements>
```

This block of code literally means while the conditional clause evaluates to $True$, execute the body of statements over and over.

```
>>> def countdown(x):
...     while x > 0:
...         print(x)
...         x = x - 1
...     print("Blastoff!")
...
>>> countdown(3)
3
2
1
Blastoff!
```

1. Fill in the is_prime function to return $True$ if n is a prime and $False$ otherwise. Hint: use the $\%$ operator. $x\%y$ returns the remainder when x is divided by y.

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

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.

3.1 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):
```

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

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.

3.2 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):
```

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):
```

3.3 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.

3.4 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):
```

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):
```

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?

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

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

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

4. Draw the environment diagram for this.

```
>>> 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)
```