BASICS REVIEW! Fun

COMPUTER SCIENCE 61A

September 27, 2015

1 Functions, While Loops, if statements

1. Implement `fizzbuzz(n)`, which prints numbers from 1 to `n` (inclusive). However, for numbers divisible by 3, print “fizz”. For numbers divisible by 5, print “buzz”. For numbers divisible by both 3 and 5, print “fizzbuzz”.

   This is a standard software engineering interview question, but even though we’re barely one week into the course, we’re confident in your ability to solve it!

```
def fizzbuzz(n):
    """
    >>> result = fizzbuzz(16)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
```

```
    16
    >>> result is None
    True
    """
```

2. Fill in the `choose` function, which returns the number of ways to choose `k` items from `n` items. Mathematically, `choose(n, k)` is defined as:

$$\frac{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}{k \times (k-1) \times (k-2) \times \cdots \times 2 \times 1}$$

```python
def choose(n, k):
    """Returns the number of ways to choose K items from
       N items.

    >>> choose(5, 2)
    10
    >>> choose(20, 6)
    38760
    """
```

2 Environment Diagrams and Lambdas Expressions!

Lambda expressions are one-line functions that specify two things: the parameters and the return expression.

A lambda expression that takes in no arguments and returns 8:

$$\texttt{lambda:}\ \underbrace{\texttt{8}}_{\text{return value}}$$

A lambda expression that takes two arguments and returns their product:

$$\texttt{lambda}\ \underbrace{\texttt{x, y}}_{\text{parameters}}\texttt{:}\ \underbrace{\texttt{x * y}}_{\text{return expression}}$$

Unlike functions created by a `def` statement, the function object that a lambda expression creates has no intrinsic name and is not bound to any variable. In fact, nothing changes in the current environment when we evaluate a lambda expression unless we do something with this expression, such as assign it to a variable or pass it as an argument to a higher order function.

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 that would result from executing the following code

```
>>> a = 5
>>> lambda1 = lambda b: b + a
>>> multiply_by = 5
>>> (lambda a: lambda1(a) * multiply_by)(multiply_by)
```

3. Write the environment diagram for the following lambda execution

```
>>> y = 4
>>> a = 2
>>> (lambda x: lambda y: x(y))(lambda a: a ** 2)(2)
```

3 Basic 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. Write a recursive function that creates a new string, but reversed. You may not use [::-1]

```
def reverse_string(string):
    """
    >>> reverse_string("Cats")
    'satC'
    >>> reverse_string("ats")
    'sta'
    """
```