Review/Exercise 4

Sources

- HKN Spring 2013 MT1 Review Slides
- HKN Spring 2013 MT2 Review Slides
- CS 61A Spring 2013 TA MT1 Review Session Slides

Environment Diagrams

1. Rule One - Global Frame: The frame in which all (python) programs begin. Draw a box and label it "Global Frame."
2. Rule Two - Assignment / Bindings
   - Variable **names** are bound to their **values** in the **current frame**
   - **Evaluate** the **right side**
     - **Bind** the variable **name** on left side to whatever the right side **evaluated to**
   - def statements and import statements are also assignments!
   - lambdas are **expressions** and only show up when bound (as return values or to names)
   - Note that for functions, sometimes you need to denote the parent frame
3. Rule Three - Variable Lookup
   - When you **evaluate** an expression and look up the **value** of a variable, start in the **current frame** and **follow the parent frames**
4. Rule Four - Function Calls
   - For a **user-defined function call**, draw a **new frame**! Note: This is the **only situation** in which you draw a new frame.

Environment Diagrams

```
the = 4
def boom(goes):
        def dynamite():
                return boom(goes-1)
        if goes < the:
                return 9
        goes += 4
        return dynamite
the = boom(5)()
boom(10)
```

Recursion

A recursive function has two important components:

- A *base case*.
- A *recursive case*.

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

Recursion

Write a recursive function eat_chocolate that takes in a number of chocolate pieces and returns a string as follows:

```
>>> eat_chocolate(5)
"nom nom nom nom nom"
>>> eat_chocolate(2)
"nom nom"
>>> eat_chocolate(1)
"nom"
>>> eat_chocolate(0)
"No chocolate :(”
```

```
def eat_chocolate(num_pieces):
    if num_pieces == 0:
        return "No chocolate :("
    elif num_pieces == 1:
        return "nom"
    return "nom " + \ eat_chocolate(num_pieces - 1)
```

Fibonacci

Write a function that prints out the first *n* fibonacci prime numbers (a number that is both a Fibonacci number and a prime number). Assume that we gave you a function is_prime that returns a boolean expressing whether or not a number is prime.

```
def nth_fib_prime(n):
    count = 0
    curr, next = 2, 3
    while count < n:
        if is_prime(curr):
            print(curr)
            count += 1
        curr, next = next, curr + next
```

Iterables

- Lists - Sequences that are **mutable.** We can add, remove, and change the items of a list.
- Tuples - Sequences that are **immutable.** We **cannot** change the items in a tuple, only create new ones.
- Dictionaries - Stores data by mapping keys to values. Remember that they are unordered and the keys are unique!
  - Remember that dictionaries have unique keys! (If I try to add a key that already exists, it overrides the previous value with the new one.)

Rlist Implementation

This is the data abstraction that we will be using for our immutable rlists:

```
empty_rlist = None

def rlist(first, rest):
    """Construct a recursive list from its first element and the rest."""
    return (first, rest)

def first(s):
    """”Return the first element of a recursive list s."""
    return s[0]

def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
def less_rlist(n, r):
    """Construct an rlist containing only values from r less than n."""
    if r == empty_rlist:
        return r
    if first(r) < n:
        return rlist(first(r), less_rlist(n, rest(r)))
    return less_rlist(n, rest(r))


def greater_rlist(n, r):
    """Construct an rlist containing only values from r greater than or
    equal to n."""
    if r == empty_rlist:
        return r
    if first(r) >= n:
        return rlist(first(r), greater_rlist(n, rest(r)))
    return greater_rlist(n, rest(r))
```

What would Python print?

```
class Animal(object):
        def __init__(self, name):
                self.n = name
                self.hunger = 0
        def eat(self, food):
                self.hunger+=1
                if self.hunger >= 2:

        print('Dead')
                else: self.hunger = 0

        print('eaten')
        def name(self):
                if self.hunger >= 3:
                        return 'Dead'
                else: return self.n
```

```
>>> c = Animal('cow')
>>> c.eat('grass')
eaten
>>> c.eat('grass')
Dead
```

General Tips

- If you don’t get it, ask.
- Don’t procrastinate on the projects!
- When preparing for the exams, do tons of practice problems, but make sure you understand them before moving on.

Final Words