## Jingchen LI / jl3426@cornell.edu

## Backend 1:

1. A class variable belongs to the class and therefore all instances instantialized can access and modify (if not a constant) it. A instance variable only beongs to a certain instance, instance varaibles belongs to two different instances are independent.
2. Bubble sort uses a moving windows size of 2 and stride of 1. At each step, it check whether two elements in the winows is sorted, if not, it swap them. Then it moves the window by 1. Repeat until in an round there is no more swapping happens.

[**“celery”, “carrot”**, “cabbage”, “tomato”] -> [ “carrot”, **“celery”, “cabbage”**, “tomato”] -> [ “carrot”, **“cabbage”**, **“celery”,** “tomato”] -> [ “carrot”, **“**cabbage”, **“celery”, “tomato”**] (already in right order)-> [ **“carrot”, “cabbage”**, “celery”, “tomato”] -> [ “cabbage”, “carrot”, **“celery”, “tomato”**] -> later on no more swap happens and Bubble sort ends

1. **Max increase**

def maxInc(array):

"""

Input an array of numbers. Return the maximum increase between a later number and a earlier number

"""

if len(array) == 0:

raise Exception("Array is empty !!!")

increased = False # whether the array has ever increased

max\_increase = 0 # initial value

min = array[0]

for num in array:

if num < min:

min = num

dif = num - min

if dif > max\_increase:

max\_increase = dif

increased = True

if not increased:

raise Exception("Array hasn't increased!!!")

return max\_increase

1. **Majority entry**

def majorityEntry\_hash(array):

"""

Given an array of objects, returnt the majorty entry

"""

# Assume the objects can be hashed

# Then can use a hash map to keep track of the counts of each number

# Use O(n) space and O(n) time

from collections import defaultdict

counts = defaultdict(int)

for item in array:

counts[item] += 1

for item, count in counts.items():

if count > len(array)/2:

return item

raise Exception("No majority item!!!")

def majorityEntry\_vote(array):

"""

I thought the question can't be that easy and googled a bit to find

Boyer-Moore Majority Vote algorithm. This is the Python implementation

"""

if len(array) == 0:

raise Exception("No majority item!!!")

candidate = array[0]

count = 1

for i in range(1, len(array)):

# Examine whether the new object should vote for the current candidate or against

if array[i] == candidate: # constant time

count += 1

else:

count -= 1

if count == 0: # if current candidate has no votes, change it to the latest item

candidate = array[i]

count = 1

# Now candidate is the one with most counts

# Need to check if it's the majority

isMajority = array.count(candidate) > len(array)/2

print(candidate, count)

print(array.count(candidate))

if isMajority:

return candidate

else:

raise Exception("No majority item!!!")

## Backend 2:

1. **Regex**

# Regex for hexidecimal numbers

import re

test\_s\_matches = ["0x0f4","0acdadecf822eeff32aca5830e438cb54aa722e3","8BADF00D"]

test\_s\_noMatches = ["ShouldntMatchThis"]

pattern = r"(?:0[xX])?[0-9a-fA-F]+"

for s in test\_s\_matches:

res = re.match(pattern, s)

if res != None:

print(res.group())

else:

print("Wrong! Didn't match {}".format(s))

for s in test\_s\_noMatches:

res = re.match(pattern, s)

if res != None:

print("Wrong! Shouldn't match {}".format(s))

1. **Two classes**

Square is a special kind of rectangular so I put it as the subclass of rectangular. It will inheritage the fields and class methods from class Rectangular. On top of these, since user can use one parameter to describe a square, I modifies the constructor a bit so that one one parameter is required. I remain the width/height field in class Square because people may still want to refer to those fields even tough they are numericall the same.

class Rect:

def \_\_init\_\_(self, width, height):

self.w = width

self.h = height

def get\_perimeter(self):

return (self.w\*2 + self.h\*2)

def get\_area(self):

return (self.w \* self.h)

class Square(Rect):

def \_\_init\_\_(self, length):

Rect.\_\_init\_\_(self, length, length)

self.length = length

1. **XML parser**

To use, please initialize a XML\_parser object and then call the self.max\_count(string).

Here, I assume an XML string contains “\n” as a line changer: "<note date='2008-01-10'>\n<book name='Red'></book>\n</note>"

class XML\_parser():

def \_\_init\_\_(self):

self.name = "I read. I parse. I count."

def count\_tags(self, string):

"""

Read a stringlized XML file and return the counts of all tags

"""

from collections import defaultdict

if len(string) == 0:

raise Exception("Input string is empty!!!")

xml = string.split("\n") # split the string into lines, each line is an root

counts = defaultdict(int) # a hashmap recording the counts for each unique tag

attributes = defaultdict(int) # a hashmap recording the number of attributes of a certain tag

for i in xrange(len(xml)): # for each line

line = xml[i]

j = 0

while line[j] != "<" and j < len(line):

j += 1

if j == len(line): # until the end still can't find a tag element, a borken line

raise Exception("Line {} is a broken line!!!".format(i))

# This line is a closing bracket </tag>, no attributes, ignore

if line[j] == "<" and line[j+1] == "/":

continue

# Have foundd the opening bracket in this line

else:

k = j+1

while line[k] != ">": # read until closing bracket

k += 1

bracket = line[j:k+1] # find the brackt

bracket = bracket.split(" ")

tag = bracket[0][1:] # first item in a bracket includes the tag

counts[tag] += 1

attributes[tag] += self.count\_attributes(bracket)

return counts, attributes

def count\_attributes(self, lst\_string):

"""

Given a splited bracket, count the occurence of attributes

"""

count = 0

for item in lst\_string:

if "=" in item:

count += 1

return count

def max\_count(self, string):

"""

Use the self.count\_tags(string) to find the tag with most counts

"""

counts, attributes = self.count\_tags(string)

if len(counts) > 0:

max\_count = 0

max\_tag = ""

for tag, count in counts.items():

if count > max\_count:

max\_count = count

max\_tag = tag

return attributes[max\_tag]

else:

raise Exception("String doesn't have tag or borken!!!")

1. **Calculate f**

# When n > 2 need to use recursion + memory

# Use an inner function to impleement recursion

def do\_f\_recur(n, memory):

"""

Helper function for do\_f(). Use recursion and memory.

"""

if n in memory:

return memory[n]

else:

return (3\*do\_f\_recur(n-1, memory) - do\_f\_recur(n-2, memory))

# Main function, only need to input n

def do\_f(n):

"""

Given a number, return value of f(n)

"""

if n < 0:

raise Exception("n must be larger or equal to 0!!!")

elif n == 0:

return 1

elif n == 1:

return 3

else:

memory = {0:1, 1:3}

return do\_f\_recur(n, memory)

# Bonus: non-recursive

def do\_f\_iter(n):

"""

Use for-loop to "grow" the value of latest i until i reaches target n

"""

if n < 0:

raise Exception("n must be larger or equal to 0!!!")

elif n == 0:

return 1

elif n == 1:

return 3

else:

previous = 3

pre\_previous = 1

for i in range(2, n+1):

num = 3\*previous - pre\_previous

pre\_previous = previous

previous = num

return num

## Backend 3: MySQL

1. **Create table**

CREATE TABLE ‘Users’

(

id INT,

lastname VARCHAR(255),

firstname VARCHAR (255),

gender VARCHAR (255),

email VARCHAR (255),

created\_at TIMESTAMP

);

1. **10 most recent added users**

SELECT \*

FROM Users as u

ORDER BY u.created\_at

Limit 10

1. **Duplicated emails**

SELECT u.email

FROM Users as u

GROUP BY u.email

Having Count(\*) > 1

1. **Duplicated emails using left join**

SELECT DISTINCT u1.email

FROM Users as u1

LEFT JOIN Users as u2

ON u1.email = u2.email AND u1.id < u2.id

## Creativity:

1. Depends on whether the blindness happens before this guy knows color. If he/she does know clolor, I just need to remind he/she: yellow is the color of warning / sun in the noon… If doesn’t I will try to assiciate yellow with warmth that he/she can feel. Like yellow is the color of fire in the fireplace.
2. Build a robot that build a robot that can only build itself. Untill the end of world or all resources on the earth is used up then they will colonize the universe to keep building themselves…
3. I have only 4 minutes left and I don’t know what’s a Battleship game. But it sounds like a class inheritance questions. So I will use chess as an example:

* Game: a game object can holds a board, and record the result of a game and has a function “play()” that reads in the input from players and call other functions to continue the game. And it must have a function to examine the result of a game.
* Board: a board objects can holds chess pieces
* Chess pieces: they implements the logic of move and attack
  + Basic pieces: super class of all pieces: can move, have status like live or dead
  + For each type of piece: have a special move function and attach function