CS 9H Final Exam Review ANSWER KEY

Python

1 Warm-up

What will Python 2 print?

a)
```
>>> (2**3, 3) + (1,2)
(8, 3, 1, 2)
```

b)
```
>>> "foo" and [] or ""
''
```

c)
```
>>> "foo" and [] or 42
42
```

d)
```
for i in "0123":  print i*2
00
11
22
33
```
*Note that these are all type `str`

e) Write a statement that returns `[1, 2, 4, 8, 16, 32]` (list of powers of 2 through 2^5)

i) using `map` and `lambda`.
```
map(lambda x:  2**x, range(6))
```

ii) using list comprehensions.
`[2**i for i in range(6)]`

f) `>>> def rearrange(arg): arg = arg[len(arg)//2:] + arg[:len(arg)//2]`
`>>> a = [2,4,6,8,10]; rearrange(arg)`

This doesn't do anything. The `arg` that the rearranged list is assigned to gets deleted when the function call terminates. If you want to modify the input, do `arg[:len(arg)] = arg[len(arg)//2:] + arg[:len(arg)//2]`

2 Collections and Mutation

What does the following method do? What is the structure of the parameter it takes? Give a general explanation, don't just describe each line of code. Provide an example input and output.

```
def wat(foo):
    ret_val = {}
    for thing in foo:
        if thing[0] not in ret_val:
            ret_val[thing[0]] = {}
        for baz in thing[1]:
            if baz not in ret_val[thing[0]]:
                ret_val[thing[0]][baz] = thing[1][baz]
            else:
                ret_val[thing[0]][baz] += thing[1][baz]
    return ret_val
```

Answer: tl;dr `wat` takes a list/tuple of lists/tuples `(label, dictionary)` and returns a dictionary that aggregates the all the dictionaries with the same label. The way I like to do this sort of problem is to draw out the data structure as I get information about it. E.g. if you know it could be a list of len-2 lists, write out `[[ , ], [ , ], [ , ]]` and fill in the rest later, instead of trying to think about the whole thing before writing it down.

```
# on the exam a simple example such as
# INPUT
# [("a":{"1":2,"2":4}), ("b":{"3":5}), ("a":{"1":4})]
# OUTPUT
# {"a": {"1":6, "2":4}, "b":{"3":5}}
```

```
# would be fine

# example usage where variable names are useful
def word_freq(docs):
    class_freqs = {}
    for doc in docs:
        if doc[0] not in class_freqs:
            # doc is the label
            class_freqs[doc[0]] = {}
        for word in doc[1]:
            if word not in class_freqs[doc[0]]:
                class_freqs[doc[0]][word] = doc[1][word]
            else:
                class_freqs[doc[0]][word] += doc[1][word]
    return class_freqs

    d1 = "hey can you check out the unix review questions thanks"
    d2 = "hot hot hot local singles in your area"
    d3 = "urgent money transfer from nigeria"
    d4 = "lololol hey kevin look at this cat video"
    d5 = "hot new job make money from home"
    d6 = "what does cat do in unix again"

    def count(doc):
        freqs = {}
        for word in doc.split():
            try:
                freqs[word] += 1
            except KeyError:
                freqs[word] = 1
        return freqs

    docs = [("ham",count(d1)), ("spam",count(d2)), \
("spam",count(d3)), ("ham",count(d4)), ("spam", count(d5)), \
("ham", count(d6))]

INPUT
[('ham', {'can': 1, 'check': 1, 'hey': 1, 'out': 1, 'questions': 1, \
'review': 1, 'thanks': 1, 'the': 1, 'unix': 1, 'you': 1}),
 ('spam', {'area': 1, 'hot': 3, 'in': 1, 'local': 1, 'singles': 1, 'your': 1}),
 ('spam', {'from': 1, 'money': 1, 'nigeria': 1, 'transfer': 1, 'urgent': 1}),
```

```
('ham', {'at': 1, 'cat': 1, 'hey': 1, 'kevin': 1, 'lololol': 1, 'look': 1, \
'this': 1, 'video': 1}),
('spam', {'from': 1, 'home': 1, 'hot': 1, 'job': 1, 'make': 1, 'money': 1, \
'new': 1}),
('ham', {'again': 1, 'cat': 1, 'do': 1, 'does': 1, 'in': 1, 'unix': 1, \
'what': 1})]

OUTPUT
{'ham': {'again': 1, 'at': 1, 'can': 1, 'cat': 2, 'check': 1, 'do': 1,
       'does': 1, 'hey': 2, 'in': 1, 'kevin': 1, 'lololol': 1, 'look': 1,
       'out': 1, 'questions': 1, 'review': 1, 'thanks': 1, 'the': 1,
        'this': 1, 'unix': 2, 'video': 1, 'what': 1, 'you': 1},
'spam': {'area': 1, 'from': 2, 'home': 1, 'hot': 4, 'in': 1, 'job': 1,
         'local': 1, 'make': 1, 'money': 2, 'new': 1, 'nigeria': 1,
         'singles': 1, 'transfer': 1, 'urgent': 1, 'your': 1}}
```

Technically you could also have the elements of the tuples have length greater than 2, but we wouldn't check anything past the element 1 anyway. The 0 element is something hashable, like a string, and the 1 element is a dictionary where the values can be added to each other.

This function could be used in a preprocessing step in creating a Naive Bayes classifier for documents, but you aren't required to know this.

`http://scikit-learn.org/stable/modules/naive_bayes.html`

3 OOP

a) Design a `Picture` class that holds a 2-dimensional collection of `Pixel` objects. Implement the following methods. The only error handling you need to worry about here is:

   - When creating `Pixel` instances, the rgb values must be numbers [0,255].
   - When cropping a `Picture`, the new dimensions must be smaller than or equal to the current dimensions (can't crop a `Picture` to be larger).

   In these cases, throw an appropriate type of `Exception` with a helpful message.

   ```
   class Pixel:
   ```

```
    def __init__(self, r, g, b):
        for val in [r, g, b]:
            if not 0<=val<=255:
                raise ValueError("RGB values must be between 0 and 255 "
                "inclusive.")
        self.r = float(r)
        self.g = float(g)
        self.b = float(b)

    def __str__(self):
        return str((self.r, self.g, self.b))

class Picture:
    filters = {}
    def __init__(self, pixel_array):
        self.pixels = pixel_array
        self.length = len(self.pixels)
        self.width = len(self.pixels[0])

    def __str__(self):
        """ Notice that this creates a list and turns that into a string only
        at the end. Strings are immutable, so if you built the return value
        directly, you would be creating and destroying the string repr
        every time you do a += operation. Although if you had a Picture that
        big you'd want to reimplement this to return a truncated representation
        anyway.
        """
        repr = []
        for row in self.pixels:
            for pixel in row:
                repr.append(str(pixel) + "  ")
            repr.append("\n")
        return "".join(repr)

    def crop(self, new_length, new_width):
        """crop the Picture to have the new dimensions"""
        if self.width < new_width or self.length < new_length:
            raise ValueError("crop dimensions must be less than or equal "
            "to current dimensions")
        self.pixels = self.pixels[:new_length]
        for i in range(new_length):
```

```
            self.pixels[i] = self.pixels[i][:new_width]
        self.width = new_width
        self.length = new_length

    def grayscale(self):
        """convert the Picture to grayscale by setting the RGB values
        of each Pixel to the average of the RGB values of the Pixel.
        Ex. if there is a pixel
        p = Pixel(100, 0, 20)
        in a Picture img, after calling img.grayscale(), the value of p
        would be equivalent to Pixel(40, 40, 40)
        """
        for i in xrange(self.length):
            for j in xrange(self.width):
                pixel = self.pixels[i][j]
                gray = (pixel.r + pixel.b + pixel.g) / 3.0
                # could also create and assign new gray pixel to
                # self.pixels[i][j]
                self.pixels[i][j].r = gray
                self.pixels[i][j].g = gray
                self.pixels[i][j].b = gray

    def add_filter(self, filter_name, filter_function):
        Picture.filters[filter_name] = filter_function
        # Must use Picture.filters, otherwise will throw NameError

    def apply_filter(self, filter):
        try:
            for i in xrange(self.length):
                for j in xrange(self.width):
                    self.pixels[i][j] = Picture.filters[filter](self.pixels[i][j])
        except KeyError:
            print "{0} not in filters. Available filters:".format(filter)
            for filter in self.filters:
                print filter
```

b) You realize that with the endless possibilities of image processing, it would be useful to let programmers to create and use their own filters. Update `Picture` to be able to store and use arbitrary filters. Assume that filters do not take any parameter. Example interaction, assuming that `sharpen` has been defined correctly earlier:

```
>>> img1 = Picture([[Pixel(100,0,20)]*4]*3)
```

```
>>> img2 = Picture([[Pixel(20,40, 60)]*5]*5)
>>> img1.add_filter("sharpen", sharpen)
>>> img2.apply_filter("sharpen") # img2 has been sharpened
```

c) What happens if you do

```
>>> pixels = [[Pixel(100,20,60)]*2]*2
>>> img1 = Picture(pixels)
>>> img2 = Picture(pixels)
>>> img1.grayscale()
>>> print img2
```

Both refer to the same set of pixels, so their contents will change in parallel.

4 So you think you know Python?

These are just for fun, and are not topics covered on the final. The first is from programmingwats.tumblr.com and the second is via Mehrdad Niknami.

```
>>> def my_append(item, lst = []):
>>>     lst.append(item)
>>>     return lst
>>> print my_append(1)
[1]
>>> print my_append(5, [3, 1, 4, 1])
[3, 1, 4, 1, 5]
>>> print my_append(1)
[1, 1]
```

Mutual default arguments are created only once.

```
>>> foo = float('nan')
>>> foo in [foo]
True
>>> float('nan') in [foo]
False
>>> foo is foo
True
>>> foo == foo
False
```

`in` uses an `is` comparison in its implementation. `nan != nan` (there are different ways things can fail to be numbers), but `foo == foo` is `True`.