6.189 Homework 3

Readings

How To Think Like A Computer Scientist: Monday - chapters 9, 10 (all); Tuesday - Chapters 12, 13, 14 (all). Tuesday's reading will really help you. If you're short on time, skim 12 & 13, and read 14 well - get at least through Section 14.6.

What to turn in

Turn in a printout of your code exercises stapled to your answers to the written exercises at 2:10 PM on Thursday, January 13th.

Exercise 3.1 – Additional List Practice

We suggest you do written exercises 3.7 - 3.9 before starting with the coding questions. Further, you'll know enough to work on OPT.1 by the end of Monday's lecture.

Note: So far we have been using `raw_input` to get user input. For the remainder of this course we will move away from this tool, instead writing functions that take in parameters as opposed to prompting the user for input. So, for this and all following problems, **do not use `raw_input` unless explicitly told to do so**.

Download and save the template `homework_3.py` from the course website. Write a function `list_intersection` that takes two lists as *parameters*. Return a list that gives the intersection of the two lists- ie, a list of elements that are common to both lists. Run the following test cases and make sure your results are the same (nb: the ordering of your outputs does *not* matter - `[3,2]` is the same as `[2,3]`):

```
>>> list_intersection([1, 3, 5], [5, 3, 1])
[1, 3, 5]
>>> list_intersection([1, 3, 6, 9], [10, 14, 3, 72, 9])
[3, 9]
>>> list_intersection([2, 3], [3,  3,  3,  2, 10])
[3, 2]
>>> list_intersection([2, 4, 6], [1, 3, 5])
[]
```

3.2 – Collision Detection of Balls

Many games have complex physics engines, and one major function of these engines is to figure out if two objects are colliding. Weirdly-shaped objects are often approximated as balls. In this problem, we will figure out if two balls are colliding. You'll need to remember how to *unpack tuples*; refer to Chapter 9.2 or ask an LA if this is confusing.

We will think in 2D to simplify things, though 3D isn't different conceptually. For calculating collision, we only care about a ball's position in space, as well as its size. We can store a ball's position with the (x, y) coordinates of its center point, and we can calculate its size if we know its radius. Thus, we represent a ball in 2D space as a tuple of `(x, y, r)`.

To figure out if two balls are colliding, we need to compute the distance between their centers, then see if this distance is less than or equal to the sum of their radii. If so, they are colliding.

In `homework_3.py`, write a function `ball_collide` that takes two balls as parameters and computes if they are colliding; your function should return a Boolean saying whether or not the balls are colliding. **Optional:** For a little extra challenge, write your function to work with balls in 3D space. How should you represent the balls? You will also need to write your own test cases - be sure to figure out any edge cases you need to test.

Exercise 3.3 – An Introduction to Dictionaries

Quick Reference `D = {}` creates an empty dictionary
`D = {key1:value1, ...}` creates a non-empty dictionary
`D[key]` returns the value thats mapped to by key. (What if there's no such key?)
`D[key] = newvalue` maps `newvalue` to `key`. Overwrites any previous value. Remember - `newvalue` can be *any* valid Python data structure.
`del D[key]` deletes the mapping with that key from `D`.
`len(D)` returns the number of entries (mappings) in `D`.
`x in D, x not in D` checks whether the key `x` is in the dictionary `D`.
`D.keys()` - returns a list of all the keys in the dictionary.
`D.values()` - returns a list of all the values in the dictionary.

For this exercise, write a dictionary that catalogs the classes you took last term - the keys should be the class number, and the values should be the title of the class.

Then, write a function `add_class` that takes 2 arguments - a class number and a description - that adds new classes to your dictionary. Use this function to add the classes you're taking next term to the dictionary.

Finally, write a function `print_classes` that takes one argument - a Course number (eg '6') - and prints out all the classes you took in that Course.

Example output:

```
>> print_classes('6')
6.189 - Introduction to Python
6.01 - Introduction to EECS
>> print_classes('9')
No Course 9 classes taken
```

For this exercise, we suggest using strings everywhere. Be sure to test with Course numbers that you both did and did not take! (if you're a Wellesley or grad student and are confused about course numbers, etc, ask an LA for explanations).

Exercise 3.4 – More About Dictionaries

In `homework_3.py`, under the Exercise 3.4 heading, notice that we've defined two lists:

```
NAMES = ['Alice', 'Bob', 'Cathy', 'Dan', 'Ed', 'Frank',
                 'Gary', 'Helen', 'Irene', 'Jack', 'Kelly', 'Larry']
AGES = [20, 21, 18, 18, 19, 20, 20, 19, 19, 19, 22, 19]
```

These lists match up, so Alice's age is 20, Bob's age is 21, and so on. Write a function `combine_lists` that combines these lists into a dictionary (*hint: what should the keys, and what should the values, of this dictionary be?*). Then, write a function `people` that takes in an age and returns the names of all the people who are that age.

Test your program's functions by running these lines (they are commented at the bottom of the code file; uncomment them to use them):

```
print 'Dan' in people(18) and 'Cathy' in people(18)
print 'Ed' in people(19) and 'Helen' in people(19) and\
       'Irene' in people(19) and 'Jack' in people(19) and 'Larry'in people(19)
print 'Alice' in people(20) and 'Frank' in people(20) and 'Gary' in people(20)
print people(21) ==   ['Bob']
print people(22) ==   ['Kelly']
print people(23) ==   []
```

All lines should print True. The last line is an "edge condition" that we're testing; your `people` function should be able to handle this condition (hint: what is this condition?) by simply returning an empty list...

Exercise 3.5 – Zeller's Algorithm, revisited

This is similar to the optional problem from Homework 1, but in this version we will be writing a function that takes parameters, and using dictionaries to facilitate "pretty printing" (where the answer is given to the user in a nice looking fashion). For the rules of the algorithm, please take a look at the description written in Homework 1 (available as a pdf online, under the Materials section of the course website).

Calling `zellers(''March'', 10, 1940)` should give the output:

```
Sunday
```

Hints:

1. Use a dictionary to map between the month and its numerical value.
2. You can use either a list or dictionary to convert the final output of the algorithm to the day of the week.
3. Make sure you handle the following three points correctly.
   - Note: If the month is January or February, then the preceding year is used for computation. This is because there was a period in history when March 1st, not January 1st, was the beginning of the year.
   - If the computed value of `R` is a negative number, add 7 to get a nonnegative number between 0 and 6.
   - You might need to use one of the following (but, maybe not): To convert the string '90' to the number 90, use `int('90')`; to convert the int 90 to the string '90', use `str(90)`.

Exercise 3.6 – Your First Class

This exercise, plus the remaining written questions (3.10 - 3.11) will be the start of our journey into object-oriented programming; we suggest you do these written exercises before tackling this question.

For this exercise, you will be coding your very first class, a Queue class. Queues are a fundamental computer science data structure. A queue is basically like a line at Disneyland - you can add elements to a queue, and they maintain a specific order. When you want to get something off the end of a queue, you get the item that has been in there the longest (this is known as 'first-in-first-out', or FIFO). You can read up on queues at Wikipedia if you'd like to learn more.

Create a new file called `queue.py` to make your Queue class. In your Queue class, you will need three methods:

- `__init__`: to initialize your Queue (think: how will you store the queue's elements? You'll need to initialize an appropriate *object attribute* in this method)
- `insert`: inserts one element in your Queue
- `remove`: removes one element from your Queue and returns it. If the queue is empty, return a message that says it is empty.

When you're done, you should test your implementation. Your results should look like this:

```
>> queue = Queue()
>> queue.insert(5)
>> queue.insert(6)
>> queue.remove()
5
>> queue.insert(7)
>> queue.remove()
6
>> queue.remove()
7
>> queue.remove()
The queue is empty
```

Be sure to handle that last case correctly - when popping from an empty Queue, print a message rather than throwing an error.

Exercise OPT.1 – Palindromes!

Write a function `is_palindrome` which takes a string as *parameter*, and returns True if the string is a palindrome (meaning it is the same forwards as backwards), and False otherwise. Save your work in `palindromes.py`. This problem is kind of tricky so feel free to ask for help; turn in any progress you make on it as well as comments explaining what does and doesn't work. For an additional challenge, try writing this as a *recursive* function.

Some useful things to remember:

- Use the function `len` to find the length of a string.
- To get just a piece of a string, use the slice operator. For example:

  ```
  astring = 'hello'
  substr  = astring[1:-1]  #sets substr to 'ell'
  ```

- You can decide for yourself whether you want your function to correctly identify palindromes that have spaces (such as 'able was i ere i saw elba') - remember the `string.join` method we saw in the hangman project?
- `string.lower` may also be a useful function.
- BE SURE TO TEST WELL! Include multiple test cases, including one where the word isn't a palindrome, but the first and last letters are equal (such as "yummy").
- It is easiest to do this with a `while` loop, although there are a few different ways of structuring such a loop. Think about what conditions need to be met for a palindrome to be true, and when you can stop testing for one (hint: the words 'ana' and 'anna' are both palindromes; when do we know to stop checking?)

Exercise OPT.2 – Stacks

This problem builds on your work in Exercise 3.6. Stacks are the opposite of queues - instead of being 'first-in-first-out', they use a 'last-in-first-out' (LIFO) strategy. Think of them like a pop-up stack of plates at a restaurant; the first plate put in will be the last one pulled out and used. Again, check out Wikipedia for a more in-depth explanation.

Make a new file `stacks.py` and implement a Stack class. It should be very similar to your Queue implementation; the three methods your class will need will be `__init__`, `push`, and `pop`.

When you're done, you should test your implementation. Your results should look like this:

```
>> stack = Stack()
>> stack.push(5)
>> stack.push(6)
>> stack.pop()
6
>> stack.push(7)
>> stack.pop()
7
>> stack.pop()
5
>> stack.pop()
The stack is empty
```