COMS 3101-3 Programming Languages – Python
Assignment 3 (25 points)
Due: 2PM, Sept 20th 2013

Each part of the assignment should be in the a single module (i.e., `hw3_part1.py, hw3_part2.py` …). Make sure to include some lines of code that tests the correctness of your implementation at the end of each module. It begins with the statement of `if __name__ == '__main__'`.

1. (5pt) (a) Extend your implementation for *`zip()`* from part 2 of assignment 2. Now, the function supports *arbitrary* number of lists (say *`n`*) of the same length as input and returns a list of *n*-tuples where each *n*-tuple is the tuple of the corresponding elements from input lists. (b) Likewise, extend your implementation for *`unzip()`* to support conversion for list of *arbitrary* number of tuples (say *n*) into *n* lists.

2. (5pt) Write a script to compute and print all prime numbers below *N*. Only using the following Python programming constructs.

   - Arithmetic (+, - …) and comparison(<, >, == …) operators
   - Function or nested function and return statements
   - *list*('[]') and *range()*
   - Functional programming operators of *map() / filter() / reduce()*

   Usage of any other programming features is not allowed. Therefore, you cannot use constructs for conditional statements *('if'*), loop statements *('for', 'while'*) or list comprehension.

3. (5pt) Create a class unique_list that extend built-in list type and override it's setter methods so that all of its elements are unique (i.e. no duplicates). The following shows sample operations.

```
ul = unique_list([1, 2, 3]) #  [1, 2, 3]
ul.append(4)   # [1, 2, 3, 4]
ul.append(1)   # [1, 2, 3, 4] – no change
ul[0] = 2      # [2, 3, 4] – no change
ul.insert(0,3) # [2, 3, 4] – no change
```

4. (10pt) This assignment uses farmer's market information (*markets.csv*) from the previous assignment again to build different data layout for market instance. The file contains fields of (in this order): *FMID*, *market name*, *web address*, *street address*, *city*, *state*, *zip code*, *x-coordinate*, and *y-coordinate*.

   (a) Create a class `FamersMarket` that represents a market entity.
   - The class defines attributes that correspond to fields from CSV file.
   - The class has a constructor method that takes a line as input parameter. It parses the line and initializes attributes. Set *FMID* as integer type and coordinate *x* and *y* as float type. Rest attributes are defined as string type.
   - Implement comparator methods `(__eq__(), __ne__(), __gt__(), __lt___(), __ge__(), __le__())`. These define object ordering based on each instance's *FMID* value.

   (b) Define a function `get_market_list(filename)` that takes the filename as its input and returns a list of all the contained farmers market instances. The function has to confirm the correctness of comparator operators by returning *sorted list* based on *FMID*. Note that *list* data type supports `sort()` method.

```
market_list = get_market_list(filename)

#should return the sorted list of FMID
map(lambda x: x.FMID, market_list)
```

   (c) Valid coordinate ranges are identified as (-40, -180) and (10, 80) for *x* and *y* coordinate respectively.
   - Define setter methods for *x, y* coordinate to `FarmersMarket` class.

```
def set_x(self, x):
      …
```

     When values to be set are not in the range, methods should generate error message by raising the exception,

```
raise Exception("Invalid x coordinate: {0}".format(x))
```

   - The above setter method cannot prevent you from bypassing range check with direct access to instance attributes i.e., *market.x, market.y.* Extend `FarmersMarket` class to apply the same range check semantic using Python *property()* [1].

[1] http://docs.python.org/2/library/functions.html#property

(d) For two coordinates of (x_0, y_0) and (x_1, y_1), we define distance metric function as follows

```
def dist((x0, y0), (x1, y1)):
      return math.sqrt(abs(x0 – x1) **2 + abs(y0 – y1) **2)
```

- Write a function `get_nearest_markets(x, y, n, market_list)` that returns a *list* that contains the *n* nearest markets from the specific location of coordinate *x* and *y*. *market_list* parameter is a *list* that provides all market instances.
  Hint: You can implement custom sorting with *list.sort()* method or using built-in *sorted()* by overriding input parameters of *key* or *cmp*[2].

(e) (Extra credit) Write a function `get_minimum_traveling_route(x, y, n_market_list)` that takes the list of *n* market instances (gained from `get_nearest_markets()`) and calculates the minimum traveling route beginning from coordinate of *(x, y).* In other words, the function should return (re-) ordered list of *n* markets minimizing the traveling distance. For instance, if `get_minimum_traveling_route(x, y [m1, m2, m3, m4])` returns `[m2, m3, m4, m1]`, the route `(x, y)` → `m2` → `m3` → `m1` makes up the minimum distance visiting all markets.

[2] https://wiki.python.org/moin/HowTo/Sorting/