

Solutions, Week 3, Day 1

1. Complete the graph class, based on adjacency lists. It should have functions for adding and deleting nodes and edges, as well as querying a node.

```
class Graph:
    def __init_ (self):
        self._nextId = 0
        self._nodeDict = {}

    def iterate (self):
        return iterOverDict (self)

    def iterateneighbours (self, node):
        return _iterOverNeighbours (self, node)

    def iterRandomly (self, node):
        return _iterRandom (self, node)

    def _getNextNodeId (self):
        self._nextId += 1
        return self._nextId

    def addNode (self):
        nodeId = self._getNextNodeId()
        self._nodeDict[nodeId] = []

    def delNode (self, node):
        del self._nodeDict[nodeId]
        for k, v in self._nodeDict:
            if node in v:
                v.remove (node):
                self._nodeDict[k] = v

    def addEdge (self, n1, n2):
        self._nodeDict[n1] = self._nodeDict[n1] + [n2]
        self._nodeDict[n2] = self._nodeDict[n2] + [n1]

    def delEdge (self, n1, n2):
        n1List = self._nodeDict[n1]
        n1List.remove(n2)
        self._nodeDict[n1] = n1List
        n2List = self._nodeDict[n2]
        n2List.remove(n1)
        self._nodeDict[n2] = n2List
```

```

def iterOverNodes (g):
    for k, v in g.nodeDict.iteritems():
        yield k

def _iterOverNeighbours (g, n):
    theNeighbours = g.nodeDict[n]
    for item in theNeighbours:
        yield item

def _iterRandom (g, n):
    from random import choice
    theNeighbours = g.nodeDict[n]
    theChoice = choice (theNeighbours)
    yield theChoice
    _iterRandom (g, theChoice)

```

2. Write a family of Matrix classes as described in the lecture: RegularMatrix, TriangularMatrix, SparseMatrix. Hint: dicts would be useful for SparseMatrix.

```

class AbstractMatrix:
    def __init__ (self):
        # don't do anything at the moment but will later
        pass

    def getValue (row, col):
        print "should override this in subclass!!!"

class RegularMatrix (AbstractMatrix):
    def __init__ (self, nrows, ncols, initial_elem=''):
        self._data = []
        for i in (range (nrows)):
            self._data.append ([initial_elem] * ncols)

    # ACCESSORS
    def countRows (self):
        return len (self._data)

    def countCols (self):
        if (self._data):
            return len (self._data[0])
        else:
            return 0

    def getValue (row, col):
        return self._data[row][col]

    # MUTATORS
    def setValue (row, col, val):
        self._data[row][col] = val

class TriangularMatrix (RegularMatrix):
    def __init__ (self, nrows, ncols, init_element=''):
        RegularMatrix.__init__ (self, nrows, ncols, init_element)

```

```

def getValue (self, row, col):
    theRealRow = max ([row, col])
    theRealCol = min ([row, col])
    return RegularMatrix.getValue (self, theRealRow, theRealCol)

def setValue (self, row, col, value):

class SparseMatrix (AbstractMatrix):
    def __init__ (self, init_elem=0):
        AbstractMatrix.__init__ (self)
        self._data = {}
        self._init_elem = init_elem

    def getValue (self, row, col):
        if (not self._data.has_key (row)):
            self._data[row] = {}
        theRow = self._data[row]
        if (not theRow._data.has_key (col)):
            theRow[col] = self._init_elem
            self._data[row] = theRow
        return theRow[col]

    def setValue (self, row, col, val):
        if (not self._data.has_key (row)):
            self._data[row] = {}
        theRow = self._data[row]
        if (not theRow._data.has_key (col)):
            theRow[col] = self._init_elem
        theRow[col] = val
        self._data[row] = theRow

```