6.00 Handout, Lecture 11
(Not intended to make sense outside of lecture)

```
class intSet(object):
    #An intSet is a set of integers
    def __init__(self):
        """Create an empty set of integers"""                                    t
        self.numBuckets = 47
        self.vals = []
        for i in range(self.numBuckets):
            self.vals.append([])

    def hashE(self, e):
        #Private function, should not be used outside of class
        return abs(e)%len(self.vals)

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        for i in self.vals[self.hashE(e)]:
            if i == e: return
        self.vals[self.hashE(e)].append(e)

    def member(self, e):
        """Assumes e is an integer
           Returns True if e is in self, and False otherwise"""
        return e in self.vals[self.hashE(e)]

    def __str__(self):
        """Returns a string representation of self"""
        elems = []
        for bucket in self.vals:
            for e in bucket: elems.append(e)
        elems.sort()
        result = ''
        for e in elems: result = result + str(e) + ','
        return '{' + result[:-1] + '}'

import datetime

class Person(object):
    import datetime
    def __init__(self, name):
        #create a person with name name
        self.name = name
        try:
            firstBlank = name.rindex(' ')
            self.lastName = name[firstBlank+1:]
        except:
            self.lastName = name
        self.birthday = None
    def getLastName(self):
        #return self's last name
        return self.lastName
    def setBirthday(self, birthDate):
        #assumes birthDate is of type datetime.date
```

```
        #sets self's birthday to birthDate
        assert type(birthDate) == datetime.date
        self.birthday = birthDate
    def getAge(self):
        #assumes that self's birthday has been set
        #returns self's current age in days
        assert self.birthday != None
        return (datetime.date.today() - self.birthday).days
    def __lt__(self, other):
        #return True if self's name is lexicographically greater
        #than other's name, and False otherwise
        if self.lastName == other.lastName:
            return self.name < other.name
        return self.lastName < other.lastName
    def __str__(self):
        #return self's name
        return self.name

class MITPerson(Person):
    nextIdNum = 0
    def __init__(self, name):
        Person.__init__(self, name)
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1
    def getIdNum(self):
        return self.idNum
    def __lt__(self, other):
        return self.idNum < other.idNum
    def isStudent(self)
        return type(self)==UG or
type(self)==G

class CourseList(object):
    def __init__(self, number):
        self.number = number
        self.students = []
    def addStudent(self, who):
        if not who.isStudent():
            raise TypeError('Not a student')
        if who in self.students:
            raise ValueError('Duplicate student')
        self.students.append(who)
    def remStudent(self, who):
        try:
            self.students.remove(who)
        except:
            print str(who) + ' not in ' + self.number
    def allStudents(self):
        for s in self.students:
            yield s
    def ugs(self):
        indx = 0
        while indx < len(self.students):
            if type(self.students[indx]) == UG:
                yield self.students[indx]
            indx += 1
```

```
class UG(MITPerson):
    def __init__(self, name):
        MITPerson.__init__(self, name)
        self.year = None
    def setYear(self, year):
        if year > 5:
            raise OverflowError('Too many')
        self.year = year
    def getYear(self):
        return self.year

class G(MITPerson):
    pass
```

6.00SC Introduction to Computer Science and Programming
Spring 2011