Question 1. [8 MARKS]

Suppose we are creating a program for online opinion polls. We need a class called **PollQuestion**, which records information about a single question on an opinion poll, including the question itself (e.g., "Is CSC148 the best course ever?!") and the responses that have been received. Respondents are identified by their email address, and only their first valid response is counted.

Write class **PollQuestion**. Include these methods, and no others:

- An `__init__` method.
- A method for recording a response to the poll question and the email address of the person giving the response. The only valid responses are 'yes', 'no', and 'maybe'; ignore any other responses. In addition, if a person has previously given a valid response, ignore the new one.
  Hint: Use a list to remember who has already given a valid response, and a dictionary to remember the number of people who have given each valid response.
- An `__eq__` method that reports whether one **PollQuestion** is equivalent to another object. In order for two **PollQuestion** objects to be equivalent, all of their attributes must be equivalent.

Write docstrings for the class and the methods, but **you are not required to write doctest examples**. Do not write any code that uses the class, just the class itself.

Sample solution:

```
class Poll:
    """
    An opinion poll, with the following attributes:
    question: str -- The poll question.
    respondents: list of str -- The email addresses of people who have responded.
    counts: {str -> int} -- for each valid response, the number of respondents.
              who have given that response.
    """

    def __init__(self, question):
        self.question = question
        self.respondents = []
        self.counts = {'yes': 0, 'no': 0, 'maybe': 0}

    def record_response(self, email_address, response):
        """
        (Poll, str, str) -> NoneType

        Record the response of the person with email_address to poll self. If
        they have previously responded, ignore their new response. The legal
        responses are 'yes', 'no', and 'maybe'. Do nothing if the response is
        illegal.

        >>> p = Poll('Is CSC148 your favourite course ever?')
        >>> p.record_response('Mariano', 'yes')
        >>> p.counts == {'no': 0, 'yes': 1, 'maybe': 0}
        True
        >>> p.record_response('Mariano', 'maybe')
        >>> p.counts == {'no': 0, 'yes': 1, 'maybe': 0}
        True
        """

        if (response in ['yes', 'no', 'maybe']
            and email_address not in self.respondents):
            self.counts[response] += 1
            self.respondents.append(email_address)

    def __eq__(self, other):
        ''' (Poll, object) -> bool

        Return whether self is equivalent to other.

        >>> Poll('Do you know the muffin man?').__eq__('yes')
```

```
        False
        >>> p1 = Poll('Is cake too expensive?')
        >>> p1.record_response('Marie Antoinette', 'no')
        >>> p2 = Poll('Is cake way too expensive?')
        >>> p2.record_response('Marie Antoinette', 'no')
        >>> p1 == p2
        False
        '''
        return (isinstance(other, Poll) and
                self.question == other.question and
                self.responses == other.responses and
                self.count == other.counts)


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Question 2. [7 MARKS]

Read the definition of **mystery** below. We realize it has a very inadequate docstring.

```
def mystery(L):
    ''' (object) -> int
    '''
    if isinstance(L, list):
        return len(L) + sum([mystery(x) for x in L])
    else:
        return 0
```

Trace the following calls to **mystery**. Be sure to trace them **in the order below**. Wherever you have a recursive sub-call to **mystery**, you will have already traced a call on a non-list, or a list of the same depth; **do not** expand further, but replace the call with the value you know it produces.

(a) Trace **mystery(37)**

```
mystery(37) -->
```

(b) Trace **mystery([1, 3, 5])**

```
mystery([1, 3, 5]) -->
```

(c) Trace **mystery([1, [3, 5], 7])**

```
mystery([1, [3, 5], 7]) -->
```

(d) Trace **mystery([1, [3, 5, [7, 9]], 11])**

```
mystery([1, [3, 5, [7, 9]], 11]) -->
```

Sample solution:

(a) Trace **mystery(37)**

```
mystery(37) --> 0
```

(b) Trace **mystery([1, 3, 5])**

```
mystery([1, 3, 5]) --> 3 + sum( [mystery(1), mystery(3), mystery(5)] )
                   --> 3 + sum( [0, 0, 0] )
                   --> 3
```

(c) Trace **mystery([1, [3, 5], 7])**

```
mystery([1, [3, 5], 7]) --> 3 + sum( [mystery(1), mystery([3, 5]), mystery(7)] )
                  --> 3 + sum( [0, 2, 0] )
                  --> 5
```

(d) Trace **mystery([1, [3, 5, [7, 9]], 11])**

```
mystery([1, [3, 5, [7, 9]], 11])
                  --> 3 + sum( [mystery(1), mystery([3, 5, [7, 9]]), mystery(11)] )
                  --> 3 + sum( [0, 5, 0] )
                  --> 8
```

Question 3. [10 MARKS]

Suppose we are creating a program for student records. We will need three classes to represent students: a general student class and specific subclasses for undergrad students and grad students. In this question, you will write classes **Student** and **Undergrad**, *but not* class **GradStudent**. We're telling you about grad students so you can envision the overall design we have in mind.

Here is what the classes must implement:

- All students have a name and a student number. We also record the number of credits the student has. We won't record any other information, not even *which* courses they have taken.
- In addition, undergrad students have a major, and grad students have an advisor and a program (*e.g.*, 'MSc' or 'PhD').
- For all students, we need a method to call when the student completes a course. It will have a parameter for the grade. For undergrads, a grade of 50 or higher is a pass and earns one credit; for grad students, 70 or higher is a pass and earns one credit.
- All students need a method that indicates whether or not the student can graduate. Undergrads can graduate if they have completed 20 courses. Grad students have different requirements.

On the next two pages, complete the code we have begun for the general class **Student**, and write the specific subclass **Undergrad**. **(Do not write the class for grad students.)** Include only the methods described above, plus an `__init__` method wherever appropriate. (Don't write `__eq__`, `__str__`, or `__repr__` methods.) Write docstrings for the class and the methods, but **you are not required to write doctest examples**.

Turn the page to add your code to what we have begun.

```
class Student:
    '''A student with these attributes:




    This is an abstract class.  Only a child class should be instantiated.
    '''

    def __init__(self, name, student_number):
        '''
        (Student, str, int) -> NoneType

        Initialize this student (self) with name, student_number, and
        zero credits so far.
        '''







    def complete_course(self, grade):
        '''
        (Student, int) -> NoneType

        This student (self) has completed a course with this grade.
        Update this student (self) if grade is high enough.
        '''







    def can_graduate(self):
        '''
        (Student) -> bool

        Return whether or not this student (self) satisfies the graduation
        requirements.
        '''
```

```
class UndergradStudent(Student):
```

Sample solution:

```
class Student:
    '''A student with these attributes:
    name: str -- the name of the student
    student_number: int -- student number
    num_credits: int -- the number of credits the student has earned.

    This is an abstract class.  Only a child class should be instantiated.
    '''

    def __init__(self, name, student_number):
        '''
        (Student, str, int) -> NoneType

        Initialize this student (self) with name, student_number, and
        zero credits so far.
        '''
        self.name, self.student_number = name, student_number
        self.num_credits = 0

    def complete_course(self, grade):
        '''
        (Student, int) -> NoneType

        This student (self) has completed a course with this grade.
        Update this student (self) if grade is high enough.
        '''
        raise(NotImplementedError)

    def can_graduate(self):
        '''
        (Student) -> bool

        Return whether or not this student (self) satisfies the graduation
        requirements.
        '''
        raise(NotImplementedError)

class UndergradStudent(Student):
    ''' An undergraduate student with these additional attributes:
    major: str -- the student's major
    '''

    def __init__(self, name, student_number, major):
```

```
        ''' (UndergradStudent, str, int, str) -> NoneType

        Initialize this undergrad student (self) with name, student_number,
        zero credits so far, and major.
        '''
        Student.__init__(self, name, student_number)
        self.major = major

    def complete_course(self, grade):
        ''' (UndergradStudent, int) -> NoneType

        This student (self) has completed a course with this grade.
        Update this student (self) if grade is high enough.

        >>> ug = UndergradStudent('Fred', 1, 'paleontology')
        >>> ug.complete_course(86)
        >>> ug.num_credits
        1
        >>> ug.complete_course(30)
        >>> ug.num_credits
        1
        '''
        if grade >= 50:
            self.num_credits += 1

    def can_graduate(self):
        ''' (UndergradStudent) -> bool

        Return whether or not this undergrad student (self) satisfies
        the graduation requirements.

        >>> ug = UndergradStudent('Fred', 1, 'paleontology')
        >>> ug.can_graduate()
        False
        >>> ug.num_credits = 20
        >>> ug.can_graduate()
        True
        '''
        return self.num_credits >= 20

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```