Question 1. [10 MARKS]

Implement a class that models a sales representative who earns commission. The sales rep has a percentage comission earned on sales, plus a bonus of $10 on any sale that exceeds $100. Your class implementation should include only the following (these are the only parts we will grade):

- a declaration of class name, and a class docstring
- an _init_ method
- a method to make a sale and update the total sales by this rep, and the combined total of commission and bonuses
- a method to report the total sales and earnings of this rep
- an _eq_ method to report whether the attributes of one sales representative are equivalent to those of another

All methods must have proper docstrings, except no examples are required.

```
class SalesRepresentative:
    """
    A sales representative.
    === Attributes ===
    @param float commission: commission rate on sales
    """
    # non-public attributes
    # @param float _total_sales: total sales by this SalesRepresentative
    # @param float _earnings: earnings of this SalesRepresentative
    def __init__(self, commission):
        """
        Create SalesRepresentative self with rate commission on sales.

        @param SalesRepresentative self:
        @param float commission: rate of commission earned on sales
        @rtype: None
        """
        self._total_sales, self._earnings = 0.0, 0.0
        self.commission = commission

    def make_sale(self, price):
        """
        Update self for making a sale of price dollars.

        @param SalesRepresentative self:
        @param float price: amount, in dollars, of sale
        @rtype: None
        """
        self._total_sales += price
        self._earnings += price * self.commission
        if price > 100.00:
            self._earnings += 10.0

    def report(self):
        """
        Report total sales and earnings

        @param SalesRepresentative self:
        @rtype: str
        """
        return "Sales: {}, Earnings: {}".format(self._total_sales,
                                                self._earnings)

    def __eq__(self, other):
        """
        Return whether SalesRepresentative self is equivalent to other.

        @param SalesRepresentative self: this sales representative
        @param SalesRepresentative|object other:
        @rtype: bool
        """
        return (type(self) == type(other) and
                self.commission == other.commission and
                self._earnings == other._earnings and
                self._total_sales == other._total_sales)
```

Question 2. [10 marks]

Implement a class that models a quiz question. A quiz question provides the question text, and a user is able to enter a response to that text. Once a response is entered, a quiz question reports whether the response is correct or not, by comparing it to the correct answer.

Also implement two subclasses to model multiple-choice quiz questions, and numerical quiz questions. Multiple choice quiz questions accept responses that are one of: "a", "b", "c", "d", or "e", and the correct answer must be one of these. Numerical quiz questions accept responses that are floats, and a correct answer is one that is in a given range, for example (0.99, 1.01).

Your design of these classes should aim to minimize duplicate code, except that **all** methods that are defined in the subclasses should also be defined in the superclass (although perhaps not implemented). You should write docstrings for each class and method.

Indicate which methods are inherited, overridden, or extended, with a brief comment explaining why you chose each approach (inherited, overridden, or extended) for these two subclasses.

For this question, we do **not** require **_str_** or **_eq_** methods.

```
class QuizQuestion:
    """
    A question on a quiz.
    === Attributes ===
    @param str text: text of this quiz question
    """
    def __init__(self, text):
        """
        Create a new QuizQuestion self with text
        and a correct_answer.

        @param QuizQuestion self:
        @param str text: text of question
        @rtype: None
        """
        self.text = text

    def check_response(self, response):
        """
        Check whether user response to text of question is correct.

        @param QuizQuestion self:
        @param str response: response to question
        @rtype: bool
        """
        raise NotImplementedError("subclass this")


class NumericalQuizQuestion(QuizQuestion):
    """
    A numerical quiz with floating-point answer
    """
    # non-public Attribute
    # @param tuple[float] correct_answer: range for correct answer
    def __init__(self, text, correct_answer):
        """
        Create a NumericalQuizQuestion expecting a correct float
        within range correct_answer.
        Extends QuizQuestion.__init__(self)

        @param NumericalQuizQuestion self:
        @param tuple[float] correct_answer:
        @rtype: None
        """
        super().__init__(text)
```

cont'd...

```
        self._correct_answer = correct_answer

    def check_response(self, response):
        """
        Report whether reponse is correct according to
        self._correct_answer
        Overrides QuizQuestion.check_response

        @param NumericalQuizQuestion self:
        @param str response: str[float] answer to this question
        @rtype: bool
        """
        return (self._correct_answer[0] < float(response) <
                self._correct_answer[1])


class MultipleChoiceQuizQuestion(QuizQuestion):
    """
    A multiple choice quiz question with response in range "a"--"e"
    """
    # non-public attributes
    # @param str _correct_answer: one of "a", "b", ..., "e"
    def __init__(self, text, correct_answer):
        """
        Create a multiple-choice quiz question with text and
        correct_answer.
        Extends QuizQuestion.__init__(self)

        @param MultipleChoiceQuizQuestion self:
        @param str text: text of this question
        @param str correct_answer: one of "a", ..., "e"
        @rtype: None
        """
        super().__init__(text)
        self._correct_answer = correct_answer

    def check_response(self, response):
        """
        Return whether response is the correct choice among
        "a", "b", ..., "d"
        Overrides QuizQuestion.check_response

        @param MultipleChoiceQuizQuestion self:
        @param str response: one of "a", ..., "e"
        @rtype: bool
        """
        return response == self._correct_answer

    # get_response is overridden to deal with different question types
    # text easily inherited
    # __init__ is extended to store different correct_answers.
```

Question 3. [8 MARKS]

Read over the definition of **search_stack** below, then complete its implementation. Your function implementation may create as extra many instances of class Stack as you like (**hint**: this is a good idea), but the **only** methods of Stack you may use are:

add(obj) add **obj** to the top of this Stack

remove() remove and return top element of this Stack

is_empty() return whether this Stack is empty

You **may not** use any Python lists, tuples, dictionaries, or other sequence classes. You may create variables to represent ordinary Python objects, such as ints.

```
def search_stack(s, obj):
    """
    Return whether Stack s contains an object equivalent to obj.
    Restore s to the same state it started in.

    @param Stack s: the Stack to check
    @param object obj: object to search stack for.
    @rtype: bool

    >>> s1 = Stack()
    >>> s1.add(3)
    >>> s1.add(7)
    >>> s1.add(11)
    >>> search_stack(s1, 7)
    True
    >>> search_stack(s1, 9)
    False
    """

    return_flag = False
    s_tmp = Stack()
    while not s.is_empty():
        tmp = s.remove()
        return_flag = return_flag or tmp == obj
        s_tmp.add(tmp)
    while not s_tmp.is_empty():
        s.add(s_tmp.remove())
    return return_flag
```