## Question 1. [10 MARKS]

Implement a class that models a bank account. This account will know how to deposit money, withdraw money, and report the current balance in the account. Your class implementation need include only the following (the only parts we will grade):

- a declaration of class name, and a class docstring
- an __init__ method that sets up the account with an initial balance
- a method to make a deposit
- a method to make a withdrawal, charging a $10 overdraft **and** refusing the withdrawal if there are not enough funds
- a method to report the current balance

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



```
class BankAccount:
    """
    Models a Monthly Bank Account
    """
    def __init__(self, balance=0):
        """
        Instance of a Bank Account

        @param: real balance: balance
        @rtype: None
        """
        self._balance = balance
        self._intRate = 0.0125

    def make_deposit(self, deposit_amt):
        """
        Make a deposit

        @param real deposit_amt: Amount to be deposited
        @rtype: None
        """
        self._balance += deposit_amt

    def make_withdrawal(self, withdraw_amt):
        """
        Make a withdrawal

        @param real withdraw_amt: Amount to be withdrawn
```

```
        @rtype: None
        """
        if self._balance < withdraw_amt:
            self._balance -= 10
            raise ValueError("Not Enough Funds")
        else:
            self._balance -= withdraw_amt
```

## 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)
        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 docstring of **bottom_stack** below, then complete its implementation. Your function implementation may create as many extra 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 bottom_stack(s):
    """
    Return the bottom element of Stack s, or None if s
    is empty.  Restore s to the same state it started in.

    @param Stack s: Stack to get to the bottom of
    @rtype: object|None

    >>> s1 = Stack()
    >>> s1.add("one")
    >>> s1.add("two")
    >>> bottom_stack(s1)
    'one'
    """

    s_tmp = Stack()
    el = None
    while not s.is_empty():
        el = s.remove()
        s_tmp.add(el)
    while not s_tmp.is_empty():
        s.add(s_tmp.remove())
    return el
```