Question 1. [10 MARKS]

Implement a class that models a cash register in a store. This cash register will know what the HST tax rate is (charged on all sales, for simplicity), is able to make sales, and keeps track of cash received as sales and tax. 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 for a given price and amount of cash paid by the customer, recording the money paid (including tax, which this method calculates), and returning the amount of change owed
- a method to report the total number of sales made and the total cash received (including tax)
- an **__eq__** method to report whether the attributes of one cash register are equivalent to those of another cash register

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

```
class CashRegister:
"""
A Cash Register
"""
def __init__(self):
"""
Create an instance of a CashRegister.

@param CashRegister self: this CashRegister
@param: CashRegister self: this CashRegister
@rtype: None
"""
self._total_sale, self._total_cash, self._hst = 0, 0.0, 1.13

def sale_transaction(self, price, cash_given):
"""
Return change due after paying cash_given for an item worth price
plus HST.  Keep track of the number of transactions and the total
cash received, both price and tax

@param CashRegister self: this CashRegister
@param float price: Price of purchase
@param float cash_given: Cash given by customer
@rtype: float
"""
self._total_cash += price * self._hst
```

```
self._total_sale += 1
return cash_given - price * self._hst

def report_total(self):
"""
Return the number of sales and the total cash received.

@param CashRegister self: this CashRegister
@rtype: str
"""
return str(self._total_sale), str(self._total_cash)
```

class name and docstring: 2 marks

- class CashRegister: (or something) 1 mark
- description of this class 1 mark

init method: 2 marks

- docstring, including type contract 1 mark
- assigns reasonable attributes, possibly non-public

sale_transaction method: 2 marks

- docstring, including type contract (perhaps without self) 1 mark
- implementation returns change, stores cash received plus cash 1 mark

report_total method: 2 marks

- docstring, including type contract (perhaps without self) 1 mark
- implementation returns some representation of the number of sales and the cash received 1 mark

__eq__method: 2 marks

- docstring, including type contract (maybe CashRegister—object for other, or possible just CashRegister)
- implementation compares attributes of self and other

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: 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.
```

init: QuizQuestion._init_probably extended, possibly inherited, twice: 4 marks, type contract and implementation

text: inherited as an attribute or method: 1 mark, docstring and implementation

check_response: some way to handle different question types: 4 marks, type contract and implementation

reasoning: docstrings indicate inheritance, extension, overriding, implementation comments for reasons: 2 mark

Question 3. [8 MARKS]

Read over the definition of **count_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 count_stack(s):
    """
    Return the number of elements in Stack s.  Restore
    s to the same state it started in.

    @param Stack s:
```

```
    @rtype: int

    >>> s1 = Stack()
    >>> s1.add("how")
    >>> s1.add("now")
    >>> count_stack(s1)
    2
    """

    s_tmp = Stack()
    counter = 0
    while not s.is_empty():
        s_tmp.add(s.remove())
        counter += 1
    while not s_tmp.is_empty():
        s.add(s_tmp.remove())
    return counter
```

loops through given stack: 3 marks

- uses **is_empty** appropriately
- updates counter
- uses **remove** appropriately

extra stack storage: 2 marks

- initializes temporary stack
- adds/removes properly

restores original stack: 2 marks

- removes/adds properly
- uses **is_empty**

returns count: 1 mark

- after all stacking/restacking done

possible messes: separate grading scheme

- uses assumed non-public attribute to find count directly 0/8
- stores elements of s in a forbidden list, tuple, ... max 4/8