UNIVERSITY OF TORONTO
Faculty of Arts and Science

CSC148H1Y: Term Test #1

Date: Thursday June 7th, 6:10 PM - 7:00PM

Duration: 50 minutes

Instructor: Sophia Huynh

No aids allowed.

LAST NAME:
FIRST NAME:
UTORID:
UOFT EMAIL:

Please read the following guidelines carefully:

● Write your name, utorid, and email clearly on this test.
● This test has 3 questions in total, with 16 pages.
● Answer the questions clearly and completely.
  ○ If you need to use an extra page for an answer, clearly write "See Page ##" and refer to the page that you used.
● You will receive 20% of the marks for any question you leave blank or indicate "I cannot answer this question".

QUESTION 1    _____ / 10
QUESTION 2    _____ / 8
QUESTION 3    _____ / 7
TOTAL        _____ / 25
QUESTION 1. Class Design (10 marks)

On the next page, we have an implementation of class `Ticket`, which represents a movie ticket. On the following pages, implement two subclasses:

**StudentTicket**: which also has a **student number**.
The price returned by `get_price()` is $0.9 \times$ the original price passed in.

The `__str__` for a `StudentTicket` should return something in the format

```
movie (STUDENT#student number: discounted price - ORIGINAL: original price)
```

For example, for a `StudentTicket` declared with `StudentTicket('Jaws', 10.00, 12345)` where 'Jaws' is the name of the movie, 10.00 is the price, and 12345 is the student number, the `__str__` should look similar to:

```
Jaws (STUDENT#12345: 9.00 - ORIGINAL: 10.00)
```

**AdultTicket**: which has no discounts.
The `__str__` for an `AdultTicket` is in the same format as a normal `Ticket`.

You do **not** have to worry about the decimal formatting of prices.

You do **not** have to implement any methods aside from those listed (i.e. you don't need `__eq__`, `__repr__`, or anything else).

You must write docstrings for each class and method with type signatures/annotations for parameters, return types, and public attributes in the format of the code below. No docstring examples are needed.

- If your docstring is the same as the superclass, you can just write **SEE SUPER**
- If there's only minor change, you can include the change (e.g. **SEE SUPER. Ticket -> AdultTicket**)

You **must not** rewrite any methods that are simply inherited from the superclass.

For each method in the subclasses you implement: add the word "**EXTENDS**" to the end of the method docstring if it extends the parent's method or "**OVERRIDES**" if it overrides (but does not extend) the superclass' method.
class Ticket:
    """
    Ticket class.
    """
    movie: str
    """
    movie - movie name
    """
    def __init__(self, movie: str, price: float) -> None:
        """
        Initialize this Ticket with the movie movie and price price.
        """
        self.movie = movie
        self._price = price
    
def get_price(self) -> float:
        """
        Return the price of this Ticket (after any discounts are applied).
        """
        return self._price
    
def __str__(self) -> str:
        """
        Return the string representation of this Ticket.
        """
        return "{} - {:.2f}".format(self.movie, self._price)

Implement StudentTicket, and AdultTicket below, using the next pages if necessary.
Extra page for Question 1.
Extra page for Question 1.
QUESTION 2. Stacks and Queues (8 marks)

RESTRICTIONS

For all parts of Question 2, the following restrictions are in place:

- You may only use the add(), remove(), and is_empty() methods of Containers (Stacks and Queues). You do not have __str__(), __eq__().
- You may not use type(), isinstance() or any other way of checking the type of c.
- c is either a Stack or a Queue.
- You may use other data structures (lists, dictionaries, Stacks, Queues) if you would like.
- You may not access any private attributes or assume anything about how q or c are stored.
  - In other words, don't try to use or access self._content anywhere.
PART A. empty_container (2 marks)

Read the description and docstring of the function empty_container below and implement its body.

def empty_container(c: Container) -> list:
    """
    Remove all the items in c and return a list of those items.
    """
    >>> s = Stack()
    >>> s.add(1)
    >>> s.add(2)
    >>> empty_container(s)
    [2, 1]
    >>> s.is_empty()
    True

    >>> q = Queue()
    >>> q.add(1)
    >>> q.add(2)
    >>> empty_container(q)
    [1, 2]
    >>> q.is_empty()
    True
    """
PART B. is_queue (3 marks)

Read the description and docstring of the function is_queue below and implement its body.

def is_queue(c: Container) -> bool:
    ""
    Return True if c is a Queue and False if c is a Stack.
    c should still be empty at the end of the function.

    Precondition: c.is_empty() == True

    >>> s = Stack()
    >>> is_queue(s)
    False

    >>> q = Queue()
    >>> is_queue(q)
    True
    """
PART C. add_to_container (3 marks)

Read the description and docstring of the function `add_to_container` below and implement its body.

```python
def add_to_container(c: Container, lst: list, is_queue: bool) -> None:
    
    Adds the elements of lst into c so that, when items are removed from c, they are removed in the order of lst. is_queue is a boolean that represents whether c is a Queue (is_queue is True), or a Stack (in which case, is_queue is False).

    It doesn't matter if lst is modified or not (i.e. if you change lst, you don't have to put it back in order).

    Precondition: c.is_empty() == True

>>> s = Stack()
>>> add_to_container(s, [1, 2, 3], False)
>>> s.remove()  
1
>>> q = Queue()
>>> add_to_container(q, [1, 2, 3], True)
>>> q.remove()  
1
```

"""
QUESTION 3. Linked Lists (7 marks)

Below are the definitions from lecture for LinkedList and LinkedListNode for reference.

class LinkedListNode:
    ""
    A Node to be used in a LinkedList.
    ""
    next_ - The successor to this LinkedListNode
    value - The data represented by this LinkedListNode.
    ""
    next_: Union["LinkedListNode", None]
    value: object

def __init__(self, value: object, next: Union["LinkedListNode", None] = None) -> None:
    ""
    Initialize this LinkedListNode with the value value and successor next.
    ""
    >>> LinkedListNode(3).value
    3
    >>> LinkedListNode(3).next_ == None
    True
    ""
    self.value = value
    self.next_ = next

def __str__(self) -> str:
    ""
    Return a string representation of this LinkedListNode.
    ""
    >>> print(LinkedListNode(3))
    3 ->
    return "{} -> ".format(self.value)

class LinkedList:
    ""
    Collection of LinkedListNodes.
    ""
    front - first node of this LinkedList
    back - last node of this LinkedList
    size - the number of nodes in this LinkedList (>= 0)
    ""
    front: Union[LinkedListNode, None]
    back: Union[LinkedListNode, None]
    size: int
def __init__(self) -> None:
    ""
    Initialize an empty LinkedList.
    ""
    >>> lnk = LinkedList()
    >>> lnk.size
    0
    ""
    self.front = None
    self.back = None
    self.size = 0

def prepend(self, value: Any) -> None:
    ""
    Insert value to the start of this LinkedList (before self.front).
    ""
    >>> lnk = LinkedList()
    >>> lnk.prepend(0)
    >>> lnk.prepend(1)
    >>> print(lnk)
    1 -> 0 -> |
    ""
    self.front = LinkedListNode(value, self.front)
    if self.back is None:
        self.back = self.front
    self.size += 1

def __str__(self) -> str:
    ""
    Return a string representation of this LinkedList.
    ""
    >>> lnk = LinkedList()
    >>> lnk.prepend(0)
    >>> lnk.prepend(1)
    >>> print(lnk)
    1 -> 0 -> |
    ""
    cur_node = self.front
    result = ''
    while cur_node is not None:
        result += str(cur_node)
        cur_node = cur_node.next_
    return result + '|'
Read the description and docstring examples of the LinkedList method `add_after_each()` below.

```python
def add_after_each(self, new_val: Any) -> None:
    ""
    Modify this LinkedList so that it adds a new linked list node with
    the value new_val after every linked list node in lnk. Update the
    attributes of lnk as needed.
    ""
    >>> lnk = LinkedList()
    >>> lnk.add_after_each("A")
    >>> print(lnk)
    |    
    >>> lnk.prepend('H')
    >>> lnk.prepend('A')
    >>> print(lnk)
    H -> A -> |
    >>> lnk.add_after_each("A")
    >>> print(lnk)
    H -> A -> A -> A -> |
    """
```

PART A. Draw the LinkedList (2 marks)

Suppose we have a LinkedList named `hr_lnk` which looks like this:

```
<table>
<thead>
<tr>
<th>Front</th>
<th>Back</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>R</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Suppose we called `hr_lnk.add_after_each("E")`. Draw how `hr_lnk` looks afterwards, labelling the front, back, and size of the linked list.
PART B. Implement add_after_each (5 marks)

Implement the body of add_after_each below. Update the attributes of self as needed.

def add_after_each(self, new_val: Any) -> None:
    """
    Modify this LinkedList so that it adds a new linked list node with the
    value new_val after every linked list node in lnk. Update the attributes
    of lnk as needed.
    
    >>> lnk = LinkedList()
    >>> lnk.add_after_each("A")
    >>> print(lnk)
    |
    
    >>> lnk.prepend('H')
    >>> lnk.prepend('A')
    >>> print(lnk)
    H -> A -> |
    
    >>> lnk.add_after_each("A")
    >>> print(lnk)
    H -> A -> A -> A -> |
    """
Use this page for rough work. If you want work on this page to be marked, please indicate this at the location of the original question (e.g. leave a comment like "See Page ##").
Use this page for rough work. If you want work on this page to be marked, please indicate this at the location of the original question (e.g. leave a comment like "See Page ##").
Built-in Functions

len(x) -> int
Return the length of the list, tuple, dict, or string x.

max(L) -> value
Return the largest value in L.

min(L) -> value
Return the smallest value in L.

range([start], stop, [step]) -> list of integers
Return a list containing the integers starting with start and
eading with stop - 1 with step specifying the amount to increment
(or decrement). If start is not specified, the list starts at 0.
If step is not specified, the values are incremented by 1.

sum(L) -> number
Returns the sum of the numbers in L.

list

x in L -> bool
Return True if x is in L and False otherwise.

L.append(x) -> None
Append x to the end of list L.

L1.extend(L2) -> None
Append the items in list L2 to the end of list L1.

L.index(value) -> int
Return the lowest index of value in L.

L.insert(index, x) -> None
Insert x at position index.

L.pop() -> object
Remove and return the last item from L.

L.pop(i) -> object
Remove and return L[i].

L.remove(value) -> None
Remove the first occurrence of value from L.

L.sort() -> None
Sort the list in ascending order.

dict

D[k] -> value
Return the value associated with the key k in D.

k in d -> bool
Return True if \( k \) is a key in \( D \) and False otherwise.

\[ D.get(k) \rightarrow \text{value} \]
Return \( D[k] \) if \( k \) in \( D \), otherwise return None.

\[ D.keys() \rightarrow \text{list of keys} \]
Return the keys of \( D \).

\[ D.values() \rightarrow \text{list of values} \]
Return the values associated with the keys of \( D \).

\[ D.items() \rightarrow \text{list of (key, value) pairs} \]
Return the (key, value) pairs of \( D \), as 2-tuples.

\[ \text{str} \]

\[ x \text{ in } s \rightarrow \text{bool} \]
Return True if \( x \) is in \( s \) and False otherwise.

\[ \text{str}(x) \rightarrow \text{str} \]
Convert an object into its string representation, if possible.

\[ S.count(sub[, start, end]) \rightarrow \text{int} \]
Return the number of non-overlapping occurrences of substring \( sub \) in string \( S[start:end] \). Optional arguments start and end are interpreted as in slice notation.

\[ S.find(sub[, i]) \rightarrow \text{int} \]
Return the lowest index in \( S \) (starting at \( S[i] \), if \( i \) is given) where the string \( sub \) is found or -1 if \( sub \) does not occur in \( S \).

\[ S.split([sep]) \rightarrow \text{List[str]} \]
Return a list of the words in \( S \), using string \( sep \) as the separator and any whitespace string if \( sep \) is not specified.