UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm Test Sample Solutions CSC148H1F

Duration: 110 min. Instructor(s): David Liu. **Examination Aids**: Provided aid sheet

---

Name:

Student Number:

Please read the following guidelines carefully.

- Please print your name and student number on the front of the exam.
- This examination has **5** questions. There are a total of **12 pages, DOUBLE-SIDED**.
- **DO NOT** open or turn over the exam paper until the exam has started.
- You may always write helper functions unless asked not to.
- Documentation is *not* required unless asked for.
- Answer questions clearly and completely. Provide justification unless explicitly asked not to.

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

Question	Grade	Out of
Q1		9
Q2		8
Q3		9
Q4		7
Q5		7
Total		40

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

1. **[9 marks] Short answer.** You may answer the following questions in either point form or full sentences; *you do not need to write much to get full marks*!

   (a) [**1 mark**] Name two different **immutable** data types in Python.

   Solution

   `bool`, `int`, `float`, `str`, `tuple`

   (b) [**1 mark**] Name two different **mutable** data types in Python.

   Solution

   `list`, `dict`, `set`

   (c) [**1 mark**] In Python, what convention do we use to indicate that an instance attribute or method is *private*?

   Solution

   Begin the name with an underscore _.

   (d) [**1 mark**] Name one abstract class we have used in this course (e.g., from a lecture, lab, prep, or assignment).

   Solution

   `Employee`, `Vehicle`, `Player` (from lab), `PersonSprite`, `ElevatorSprite`, `ArrivalGenerator`, `MovingAlgorithm` (from Assignment 1)

   (e) [**1 mark**] Why should client code never instantiate an abstract class directly?

   Solution

   The object would have at least one method that is not implemented, and calling such a method would raise an error.

   (f) [**2 marks**] Suppose we have a variable `curr` that refers to a `_Node` in a linked list. Write a Python expression that evaluates to `True` if `curr` refers to the *second-last* node in a linked list, and `False` otherwise. (You should not assume anything about the linked list, other than `curr` refers to a node in it.)

   Solution

   ```
   curr.next is not None and curr.next.next is None
   ```

   (g) [**2 marks**] Suppose in Python we have a built-in array-based list of length 1,000,000, and a linked list of length 1,000,000. If we insert a new item at index 500,000 into each list, would it be:

   - significantly faster for the array-based list
   - significantly faster for the linked list
   - roughly the same amount of time for both lists

   Circle one of the three options, and then **explain your answer**:

Solution

This takes roughly the same amount of time for both lists. For the array-based list, the last (roughly) 500,000 elements would have to move over to make room for the new item inserted, and for the linked list, the first (roughly) 500,000 nodes would have to be traversed before reaching the place to insert the new item.

2. [**8 marks**] **Object-oriented design.** You are responsible for creating a class to represent a user in an online messaging system. In this system, every user has a username, email address, and a history of all of the messages they have received from each user, in the *reverse* order in which they were received. Here is an example of how we want to use this class.

```
>>> david = User('david123', 'david@gmail.com')
>>> diane = User('dianehorton', 'diane@gmail.com')
>>> jacqueline = User('the_chairman', 'jacqueline@gmail.com')
>>> david.message(diane, 'Hi, how are you?')  # david sends a message to diane.
>>> diane.message(david, 'I am great! How are you?')
>>> david.message(diane, 'Good---although I could use some more sleep.')
>>> diane.get_messages(david)  # The messages diane received from david, in reverse order.
['Good---although I could use some more sleep.', 'Hi, how are you?']
>>> diane.get_messages(jacqueline)
[]
```

Below and on the next page is a very incomplete class design. You have tasks marked `TODO` in the code:

(a) [**2 marks**] Document all the *instance attributes* of the `User` class. You may choose any reasonable way to store the necessary data, and may make all attributes public.

(b) [**3 marks**] Implement `User.__init__` so that it is compatible with the example code and your chosen attributes.

(c) [**3 marks**] Complete the implementations for `User.message` and `User.get_messages`.

You may assume that all usernames and email addresses are unique.

Solution

```
class User:
    """A user in an online messaging system.

    # TODO: Document all instance attributes here.
    === Attributes ===
    username: this user's username
    email: this user's email address
    messages:
        a dictionary mapping usernames to a list of messages received from
        that user.


    >>> david = User('david123', 'david@gmail.com')
    >>> diane = User('dianehorton', 'diane@gmail.com')
    >>> jacqueline = User('the_chairman', 'jacqueline@gmail.com')
    >>> david.message(diane, 'Hi, how are you?') # david sends a message to diane.
    >>> diane.message(david, 'I am great! How are you?')
    >>> david.message(diane, 'Good---although I could use some more sleep.')
    >>> diane.get_messages(david) # The messages diane received from david, in reverse order.
    ['Good---although I could use some more sleep.', 'Hi, how are you?']
    >>> diane.get_messages(jacqueline)
    []
    """
    # TODO: Write type annotations for your attributes here.
    username: str
    email: str
    messages: Dict[str, List[str]]

    # TODO: Implement User.__init__ here.
    # The method header must include a type contract, but a docstring is NOT required.
    def __init__(self, username: str, email: str) -> None:
        self.username = username
        self.email = email
        self.messages = {}

    # TODO: Implement this method.
    def message(self, recipient: User, text: str) -> None:
        if self.username in recipient.messages:
            recipient.messages[self.username].insert(0, text)
        else:
            recipient.messages[self.username] = [text]
    # TODO: Implement this method.
    def get_messages(self, sender: User) -> List[str]:
        """Return a list of the messages this user received from <sender>

        The messages should be returned in the REVERSE order from which they were received.
        """
        if sender.username in self.messages:
            return self.messages[sender.username]
        else:
            return []
```

3. [**9 marks**] **Stacks and queues.**

   (a) [**1 mark**] Here is the docstring of a function that operates on a stack. Read it and complete the doctest.

```
def keep_top(stack: Stack) -> None:
    """Remove all items except the top one from the given stack.
    Precondition: <stack> has at least one item.

    >>> s = Stack()
    >>> s.push(10)
    >>> s.push(20)
    >>> s.push(30)
    >>> keep_top(s)
    >>> s.pop()       # TODO: fill in the return value of s.pop() here.



    >>> s.is_empty()
    True
```

Solution

s.pop() returns 30.

   (b) [**2 marks**] Here is an *incorrect* implementation of this function.

```
def keep_top(stack: Stack) -> None:
    top_item = stack.pop()
    while not stack.is_empty():
        stack.pop()
        stack.push(top_item)
```

Explain: **(1)** what happens when we run the above doctest using this implementation, and **(2)** why this occurs.

Solution

(1) we get an infinite loop, because **(2)** inside the `while` loop, we both remove one item and add a new item to the stack, so its size never decreases.

   (c) [**2 marks**] Here is another *incorrect* implementation of this function.

```
def keep_top(stack: Stack) -> None:
    new_stack = Stack()
    top_item = stack.pop()
    new_stack.push(top_item)
    stack = new_stack
```

Explain: **(1)** what happens when we run the above doctest using this implementation, and **(2)** why this occurs.

Solution

(1) s.pop() returns 20 (not 30), because **(2)** the code mutates the original stack by popping off its top item (30), but then does not change the stack in any other way.

(Note that the last line `stack = new_stack` simply reassigns the variable `stack` to a new object; it doesn't mutate the original stack.)

(d) [**1 mark**] Suppose we have a `Queue` implementation that uses a Python (array-based) list, where the front of the list represents the front of the queue.

Based on this implementation, which operation do we generally expect to take *longer* (circle one):

`Queue.enqueue` `Queue.dequeue`

Explain (answers without an explanation will not receive credit):

Solution

`Queue.dequeue` takes longer because when we remove an item from the front of a Python list, all other items must be shifted over to fill in the gap left by the removal. But in `Queue.enqueue` we insert an item at the end of a Python list, *no* items must shift over.

(e) [**3 marks**] Consider the following function:

```python
def send_to_back(queue: Queue, k: int) -> None:
    """Send the first <k> items in the given queue to the end of the queue.

    Preconditions:
        k >= 1, and <queue> has at least k items
    """
    for i in range(k):
        item = queue.dequeue()
        queue.enqueue(item)
```

Suppose we use the same `Queue` implementation as described in part (d). Let n be the size of `queue`. Calculate the total number of times an item is shifted in a Python `list` when we call `send_to_back(queue, k)`, in terms of n and/or k. *Answers without an explanation will not receive credit.*

Note: We will not deduct marks for off-by-one errors here.

Solution

Inside the `for` loop, each call to `queue.dequeue` causes all other queue elements to shift over, while each call to `queue.enqueue` doesn't cause any elements to shift over.

Also, since at each iteration one `dequeue` and one `enqueue` occur, the size of the queue never changes—it's always n.

So the number of items shifted in a Python list *per loop iteration* is $n - 1$. Since there are k iterations in total, the total umber of times an item is shifted is $(n - 1) \times k$.

4. [**7 marks**] **Memory model diagrams.** Here is a short Python program.

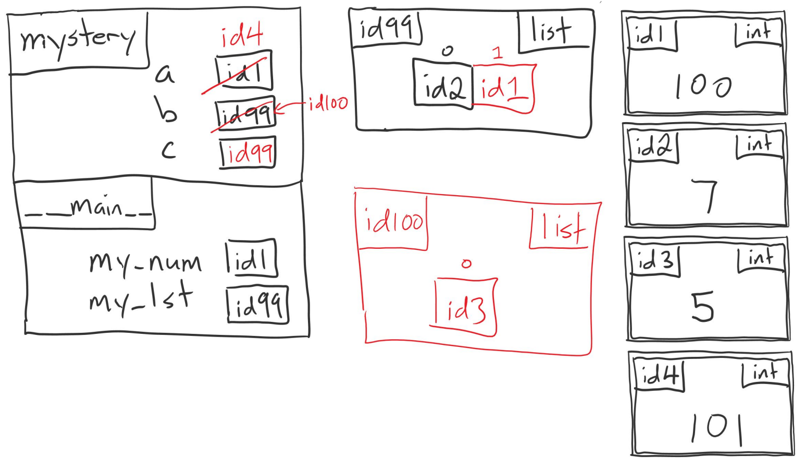
```
def mystery(a: int, b: List[int]) -> None
    c = b
    c.append(a)
    a = a + 1
    b = [5]

if __name__ == '__main__':
    my_num = 100
    my_lst = [7]
    mystery(my_num, my_lst)
```

(a) [**5 marks**] The memory model diagram below shows the state of this program's memory when `mystery` is called, but before the first line of its body has been executed.

Modify this diagram to show the state of this program's memory *immediately before the function returns* (i.e., just after executing `b = [5]`). We have provided all the `int` objects you should need for your diagram.

Solution



(b) [**2 marks**] Write down the values of `my_num` and `my_lst` *after* `mystery` returns. (We're asking for their *values*, not their ids!)

`my_num` `my_lst`

Solution

`my_num` is `100`, `my_lst` is `[7, 100]`.

5. [**7 marks**] **Linked lists.** Implement each of the following `LinkedList` methods. You may not use *any* `LinkedList` methods in your implementation; we are looking for you to work with nodes directly.

   Please refer to the provided aid sheet for documentation for the `LinkedList` and `_Node` classes.

   For each method, we have provided a part of the implementation for you already. You *must* use this as a starting point for your solution.

   (a) [**3 marks**]

Solution

```
def average(self) -> float:
    """Return the average of the numbers in this linked list.

    Preconditions:
        - this linked list is not empty
        - all items in this linked list are numbers

    >>> lst = LinkedList([10, 15])
    >>> lst.average()
    12.5
    """
    curr = self._first

    # Initialize any other variables here.
    running_sum = 0
    running_len = 0

    while curr is not None:
        running_sum += curr.item
        running_len += 1

        curr = curr.next

    # Return the average after the loop ends.
    # (You may need to do some other calculations first.)
    return running_sum / running_len
```

(b) [**4 marks**] For this method, you can, and should, create new `_Node` objects.

Solution

```
def intersperse(self, other: LinkedList) -> None:
    """Insert the items of <other> in between the items of this linked list.

    Each item in <other> is inserted immediately after the corresponding item in <self>.
    Do not mutate <other> (this includes any of its nodes).
    See the doctest below for an example.

    Precondition: <self> and <other> have the same length.

    >>> lst1 = LinkedList([1, 2, 3])
    >>> lst2 = LinkedList([10, 20, 30])
    >>> str(lst1)                        # before
    '[1 -> 2 -> 3]'
    >>> lst1.intersperse(lst2)
    >>> str(lst1)                        # after
    '[1 -> 10 -> 2 -> 20 -> 3 -> 30]'
    """
    curr1 = self._first
    curr2 = other._first

    # NOTE: You should do all of your work *inside* the while loop.
    # It is up to you to complete the while loop condition and its body.
    while curr1 is not None and curr2 is not None:
        new_node = _Node(curr2.item)
        curr1.next, new_node.next = new_node, curr1.next

        # Need to advance two nodes (since new_node was just inserted after
        # curr1).
        curr1 = curr1.next.next
        # This just needs to advance one node.
        curr2 = curr2.next
```