CSC148 Summer 2018: Midterm 1 Practice

The questions in this practice encompass more than would be on a midterm. The questions are designed to cover a breadth of topics. Within each category are multiple questions that address various topics, and then one question at the end that is more "test" styled (i.e. a question that would be fair for a test) at the end.

In addition to answering the questions within this document, you should work through the labs and past exams.

Class Design

1. **Below are the definitions for the superclass Ghost and its subclass Spectre.**

   ```python
   class Ghost:
   """
   A class representing a Ghost.
   
age - The age of the ghost
   """
   age: int
   
def __init__(self, age: int) -> None:
   """
   Initialize this Ghost with the age age.
   """
   self.age = age
   
def make_sound(self) -> None:
   """
   Make this Ghost make a sound.
   """
   print("Boo!")
   
   class Spectre(Ghost):
   """
   A class representing a Spectre.
   
age - The age of the spectre
   size - The size of the spectre
   """
   age: int
   size: int
   
def __init__(self, age: int, size: int) -> None:
   """
   Initialize this Spectre with the age age and size size.
   """
   super().__init__(age)
   ```
self.size = size

a) What happens when we run the following lines of code?
   s = Spectre(10, 3)
   s.make_sound()

b) Suppose we want to make another subclass of Ghost called Ghoul. A Ghoul has an age, as well as a name. Implement its _init_ method.

c) We want all ghosts to have a _str_ method that prints out its age in the format
   Type of Ghost (Age)

   For example, a Spectre with an age of 10 should have the str form:
   Spectre (10)

   A Ghost should not have its str method implemented (i.e. it should raise NotImplementedError).

   Write the _str_ methods for Ghost, Spectre, and Ghoul.

d) Suppose we want a Ghoul's make_sound() method to make whatever sound a Ghost does, but to also print out 'Grr...' afterwards. Implement the make_sound() method.

2. Assume the class for which the methods below are written have a super class. For each of the methods below, label whether they extend, override, or inherit the behaviour of a parent method. Docstrings have been omitted.

   a) def raise_hand(self) -> None:
      super().raise_hand()
      print("Me!")

   b) def increase_count(self) -> None:
      self.count += 1

   c) def turn_around(self) -> None:
      super().turn_around()

   d) def say_answer(self) -> None:
      print(super().say_answer + ", I think.")

3. What does it mean to extend a method?

4. What does it mean to override a method?
5. How do we define a private attribute?

6. Suppose we have a Stack, but instead of having a private _content attribute, it has a public content attribute, with content being a list of items in the Stack.

a) Suppose we have client code that creates a Stack and stores it in the variable s. Using the content attribute, how could we find the current size of the Stack?

b) Suppose we wanted to implement our Stack using a LinkedList instead of a list. Are we able to make that change? Would the client code from (a) have to change?

c) If content was a private attribute (_content) and stored as a list, how would we have to provide the size of our Stack to our client? Write code that the client would have to call in order to get the size of our Stack. The client code should not access the private _content attribute.

d) Continuing from (c), what methods within our Stack would have to change? (Code is not needed, but it would be good practice for all topics: class design, stacks, and linked lists.)

e) Continuing from (d), suppose we wanted to switch our implementation from using a list for _content to using a LinkedList. Would the client code have to change?

7. Why do we use private attributes?

8. What is encapsulation?

9. Why do we use getters and setters?

10. Below are the definitions for the class Owner and Pet.

class Pet:
    
    """
    A class representing a Pet.
    
    name - The pet's name.
    fullness - The pet's fullness
    """

    name: str
    fullness: int

    def __init__(self, name: str) -> None:
        """
        Initialize this Pet with the name name and no fullness.
        """

        >>> p = Pet("Froggy")
        >>> p.name
        'Froggy'
        >>> p.fullness
        0
class Owner:
    
    A class representing an Owner.
    
    pet - The pet of this Owner.
    name - The Owner's name.
    
    pet: Pet
    name: str

def __init__(self, name: str, pet: Pet) -> None:
    
    Initialize this Owner with the name name and pet pet.

    >>> o = Owner("Sophia", Pet("Stinky"))
    >>> o.name
    'Sophia'
    
    self.name = name
    self.pet = pet

a) Suppose we create an Owner using the code:

    o = Owner("Sophia", Pet("Stinky"))

What line of code would we have to run to get the pet's name?

b) Suppose we want a method in the Owner class called feed_pet which increases the pet's fullness by 5. Implement the method feed_pet.

c) Implement an __eq__ method for Pet. Two Pets are equal if they're both Pets, have the same name, and the same fullness.

d) Implement an __eq__ method for Owner. Two Owners are equal if they're both Owners, have the same name, and their pets are equal.

e) Implement a __str__ method for Pet. The __str__ of a pet should return a string in the form:

    Name (Fullness)

f) Implement a __str__ method for Owner. The __str__ of an owner should return a string in the form:

    Name: the str of the pet

Use the pet's str in the Owner's __str__ method.
11. When is the __str__ method used?

12. When is the __repr__ method used?

14. Describe a scenario wherein __str__ and __repr__ would logically have very different results.

15. Below is an implementation of the class Meal, which represents a meal that people can eat.

```python
class Meal:
    """
    A Meal class.
    """
    name - name of the meal
    price - price of the meal
    """
    name: str
    price: int

def __init__(self, name: str, price: int) -> None:
    """
    Initialize this Meal with the name name and price price.
    """
    self.name = name
    self.price = price

def __str__(self) -> str:
    """
    Return the string representation of this Meal.
    """
    return "{} (${})".format(self.name, self.price)

def is_healthy(self) -> bool:
    """
    Return whether this meal is healthy or not.
    """
    raise NotImplementedError

Implement the following subclasses:

- HealthyMeal: Which also has a main ingredient. is_healthy() should return True, and the __str__ for a HealthyMeal should return a string in the form:
  name ($price): main ingredient
- JunkMeal: Whose is_healthy() should return False.

Include all documentation (docstrings, type annotations, etc.) excluding docstring examples in the subclasses and in any methods you define. Additionally, mention whether a method extends or overrides the parent method in the docstring.
Stacks and Queues

1. What is a Stack?
2. What is a Queue?
3. Suppose we add "A" to a Stack. Afterwards, we add "B", and then "C". When we remove from the Stack, what do we get back?
4. Suppose we add "A" to a Queue. Afterwards, we add "B", and then "C". When we remove from the Queue, what do we get back?
5. Suppose we have a Stack that will contain only single character strings (e.g. 'A', 'b', '1'), and we want to implement it using a string as the Stack's _content. Let's call this a StringStack.

Implement the __init__(), add(), remove() and is_empty() methods for StringStack.

6. Suppose we have a Queue that will contain only single character strings (e.g. 'A', 'b', '1'), and we want to implement it using a string as the Queue's _content. Let's call this a StringQueue.

Implement the __init__(), add(), remove() and is_empty() methods for StringQueue.

7. Using only type(), is_empty(), add(), and remove(), implement the __eq__ method of a Queue. Two Queues are equal if they contain the same items in the same order. The Queue should be in its original state by the end of the method (i.e. if you remove things from the Queue, you must put everything back in the original order). Do not access _content.

8. Using only type(), is_empty(), add(), and remove(), implement the __str__ method of a Stack. The __str__ method should take the form:

Top -> items

Where items are the contents of the queue in the order they're removed.

The Stack should be in its original state by the end of the method (i.e. if you remove things from the Stack, you must put everything back in the original order). Do not access _content.

9. Using only type(), is_empty(), add(), and remove(), implement the __str__ method of a Queue. The __str__ method should take the form:
Front -> items

Where items are the contents of the queue in the order they're removed.

The Queue should be in its original state by the end of the method (i.e. if you remove things from the Queue, you must put everything back in the original order). Do not access _content.

10. Read the docstring below and implement the body of the function.

```python
def queue_to_stack(q: Queue) -> Stack:
    """
    Return a stack with the items from q. The stack returned should
    Have items removed in the same order as q.
    
    After calling this function, q should be in its original state (all
    Items in the same order).
    
    >>> q = Queue()
    >>> q.add(1)
    >>> q.add(2)
    >>> q.add(3)
    >>> s = queue_to_stack(q)
    >>> q.remove()
    1
    >>> s.remove()
    1
    >>> q.remove()
    2
    >>> s.remove()
    2
    >>> q.remove()
    3
    >>> s.remove()
    3
    """
```

Linked Lists

1. What does a LinkedListNode contain?

2. What does a LinkedList contain?

3. Suppose we have a LinkedList named lnk. How would we get the front of lnk? If lnk is an empty LinkedList, what is the front of it?

4. Suppose we have a non-empty LinkedList named lnk. How would we get the value of 2nd node in our LinkedList?
5. Suppose we have a LinkedList containing the items (from front to back) "A", "B", "C", "D".

a) Draw the LinkedList. Label the front, back, and size.

b) Suppose we want to add a new LinkedListNode with the value "E" between "B" and "C". Draw the new LinkedList, labelling the front, back, and size.

c) Continuing from (b): Which next pointers need to change?

d) Continuing from (b): How would we create our new LinkedListNode with the value "E"?

e) Continuing from (d): How would we get to the LinkedListNode with "B" as its value and set its next_pointer to refer to our node from (d)?

f) Continuing from (e): How would we update our new LinkedListNode's next_pointer to point to C?

g) Continuing from (f): How would we update the size of the LinkedList?

h) Continuing from (g): Do we need to change the front or back pointers of our LinkedList?

6. Suppose we want to implement a Stack using a LinkedList as the _content and that we'll call it LinkedListStack. The class definition and __init__ have been provided for you. Write the add(), remove() and is_empty() methods. Assume you have LinkedList defined for you already, but you only have access to the front, back, and size attributes. Assume you have LinkedListNode defined for you too, but it only contains the value and next attributes.

class LinkedListStack:
    ""
    A class representing a Stack, formed using a LinkedList.
    ""
    def __init__(self) -> None:
        ""
        Initialize an empty LinkedListStack.
        ""
        self._content = LinkedList()

        >>> s = LinkedListStack()
        >>> s.add(3)
        >>> s.add(2)
        >>> s.add(1)
        >>> s.remove()
        1
        >>> s.remove()
        2
        >>> s.remove()
        3
        ""
Below is the definition of the LinkedList method `add_before`.

```python
def add_before(self, new_value: Any, to_find: Any):
    """
    Add new_value to this LinkedList so it comes immediately before to_find.
    If to_find isn't in this LinkedList, don't modify this LinkedList.
    """
    >>> lnk = LinkedList()
    >>> lnk.prepend("H")
    >>> lnk.add_before("A", "H")
    >>> print(lnk)
    A -> H -> |
    >>> lnk.size
    2
    >>> lnk.add_before("C", "H")
    >>> print(lnk)
    A -> C -> H -> |
    >>> lnk.size
    3
    >>> lnk.add_before("O", "B")
    >>> print(lnk)
    A -> C -> H -> |
    >>> lnk.size
    3
    """
```

a) Draw a LinkedList with the items from front to back being "A", "T", "H", labelling the front, back, and size.

b) Continuing from (a): Suppose that LinkedList is named `lnk`. Suppose we call `lnk.add_before("B", "A")`. Draw the new LinkedList, labelling the front, back, and size.

c) Implement the method `add_before()`.